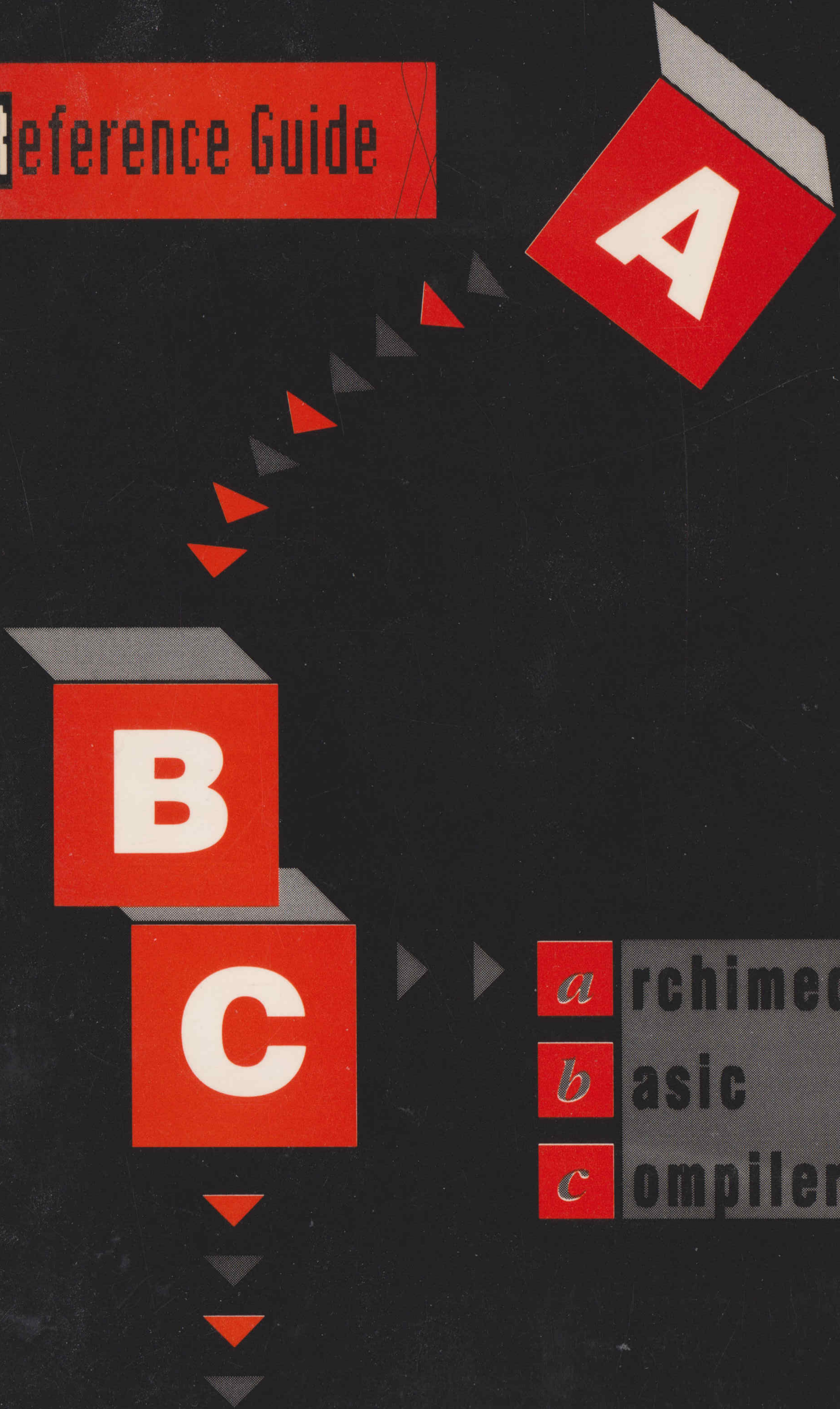
Reference Guide



archimedes basic compiler

Archimedes Basic Compiler

Reference Guide

Paul Fellows

DABS
PRESS

Archimedes Basic Compiler
Reference Guide

© Solent Computer Products Ltd.
ISBN 1-870336-76-3.

First edition November 1988

Editors: Robin Burton and Bruce Smith
Proofreading: Lynda Atherton
Typesetting: Felix Andrew and Bruce Smith
Cover: Clare Atherton

Published by Dabs Press, 5 Victoria Lane, Whitefield, Manchester M25 6AL. Telephone 061-766 8423. Telecom Gold 72:MAG11596, Prestel 942876210.

Typeset in 10 on 11pt Palatino by Dabs Press using the Acornsoft VIEW wordprocessor, MacAuthor, Apple Macintosh SE and LaserWriter II NT.

Printed and bound in the UK by Ernest Brown & Son, Sefton Street, Whitefield, Manchester. 061-796 8750

Contents

Introduction

All versions of Acorn's BBC microcomputer to date, including the Archimedes, have been supplied as standard with a version of BBC BASIC. This is an interpreter – a program which reads a BASIC program line by line, analyses and evaluates the instructions, executing them immediately. The alternative to an interpreter is a compiler. Like an interpreter a compiler also reads the whole program, but it does not execute the commands once they are identified. Instead a compiler thats the instructions of the program, usually called the *source*, and converts them into a new machine code program – called the *object* – which may then be saved for subsequent use.

Interpreters have the advantage that a program can be typed in and run straight away without any extra operations. However, they also have two major disadvantages. The first is that, especially for large programs, you can never be absolutely sure that a program is free of syntax errors. These will be found only if the interpreter tries to evaluate the instructions in which they occur. If, when a program runs, a particular path through it is not followed, possible errors in that area of the program remain undetected. Since a compiler always processes and evaluates every instruction such errors are always discovered during compilation.

The second disadvantage of interpreted programs is that their speed of execution is much slower than for compiled programs. This is because each statement must be re-interpreted every time it is executed. Under a compiled system most of this analysis is done once only – during the compilation.

The ideal is to have both an interpreter and a compiler. The interpreter is used to develop and test the program. The compiler can the fulfill two functions. It can be used initially to detect syntax errors, and finally, when development is complete, to produce the finished compiled program. The Archimedes Basic Compiler (ABC) has been produced to provide you with this ideal development environment for your Archimedes programs.

Chapter One of this Reference Guide explains how to compile BASIC programs and the various automatic processes performed by the compiler. Chapters Two to Four explain the differences between interpreted and compiled BASIC programs. Chapter Five covers the in-line assembler, while Chapter Six contains information about commands and facilities specific to controlling the compilation process.

Chapter Seven contains descriptions of the syntax of all the keywords available for use in programs to be compiled. These are listed in logical groups accordinging to function. Within each group the keywords are given in alphabetic order. For each instruction a brief description is given, followed by its syntax, details of its arguments, the effect it has and one or two examples of use.

Finally, the ABC User Guide which accompanies this Reference Guide contains more practical help and advice on using ABC. If you are using ABC for the first time you are advised to read the User Guide first of all.

1 : Compiler Operation

Getting Started

The Archimedes Basic Compiler may be used on any configuration of Archimedes computer. However the manner in which the compiler is used will ultimately depend on whether a single, dual or hard disc drive(s) are in use. Full details on how to install and use ABC can be found in the *ABC User Guide*. Please refer to this for full details and worked examples. Note that if you have a floppy disc drive, you must install ABC on a working disc (800k format) before you can use it.

Stages in Compilation

In order to compile the code the compiler must read the source file several times. By default during each read, or pass, each source line is displayed on the screen as it is processed. For most programs this won't be needed, since the compiler processes approximately 1000 lines per minute in each pass.

You can turn the listing display off for all or any part of a compilation – see 'Compiler Directives', Chapter Six for details.

The Compilation Process

During each pass through the source code, the compiler performs a different function, which ultimately results in the production of the compiled pro-gram, ie, the machine code. The separate processes are completely auto-matic, but are described here for your information.

During the first pass, the syntax of the source program is checked, and the names of all variables, procedures and functions are recorded. If the compiler encounters a syntax error (in effect any instruction it can't under-stand) the compilation terminates, with a message describing the error and an indication of its location. For example:-

```
Error: Type mismatch
50 IF A$ = 0 THEN PROCzero
```

At the end of the first pass, the compiler checks that all variables, procedures or functions used have been properly defined. If the program does contain references to undefined variables, each is presented in turn in the status window and compilation terminates.

In pass two the compiler determines the data types returned by functions. Pass two is performed, therefore, only when the program references a function for which a definition has not yet been compiled. Otherwise it moves straight on to pass three.

However, if you assign the return value of one function to be the value of a second function, defined later in the code, the compiler requires two 'second' passes. For example:

```
DEFFN1 ...... = FN2
DEFFN2 ...... = 23.5
```

In this case, during the 'first pass two' a data type cannot be assigned to FN1 since it relies on FN2, which is as yet unknown. Only later during the pass does the compiler discover that FN2 returns a real number. During the 'second pass two', therefore, the compiler knows all data types returned by all functions, and so is able to determine the data type returned by FN1.

You could minimise the need for an extra pass by taking care about the order in which your definitions are declared in the source. If the order of FN1 and FN2 were reversed in the above example the extra pass would not be needed.

During pass three the compiler carries out a pseudo-compilation, using dummy addresses for GOTOs, procedure or function calls as necessary. As pass three progresses the real address of each routine is recorded as it is encountered. By the end of the pass all addresses used in the program – both internal and external – are known, together with the total code size.

Pass four compiles the final version of the object program.

Should ABC encounter an error at any point, it will give you the option of entering directly into the Basic Editor to allow the error to be corrected.

At this point the compiled program is saved to a file, the name of which has already been specified. If no such file exists it will be created, but any existing file of this name will be overwritten. On completion, the time taken to compile the code and the size of the code produced is reported in the status window. A typical display might be:-

```
Compile time   =1614s
Codesize       =197K
```

You are then asked whether you want to run the compiled code.

Executing the Code

The compiled program is completely stand-alone and fully relocatable. No special environment or additional software (other than, where necessary, the Floating Point Emulator) is required to allow the code to be used. After successful compilation, a program can be treated just like any other machine code routine.

Interpreted vs. Compiled

In an ideal world, it would be possible to have complete source and object compatibility betweeen the BASIC interpreter and the ABC compiler. Unfortunately, this is not the case. There are certain small differences between the syntax for some instructions, as well as in the output generated. An existing program may, therefore, need some minor modification before compilation.

Some of these differences are inherent in the execution of compiled or interpreted code. Others are deliberately implemented to improve the performance of the compiled code – since the compiler is primarily intended as a development tool, not just a new way to run existing programs.

The differences between compiled and interpreted code as they affect your own programs are documented in the following chapters.

2 : Flow Control Structures

The compiler supports all the flow control structures available in BASIC V. These are:

IF ... THEN ... ELSE ... ENDIF
FOR ... TO ... STEP ... NEXT
REPEAT ... UNTIL
WHILE ... ENDWHILE
CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE
PROC and DEFPROC ... ENDPROC
FN and DEFFN ... =
ON ... GOTO
ON ... GOSUB ... RETURN
ON ... PROC

Nesting

All 'legal' nesting of structures is fully supported, to any level. For example:

```
IF A% = 1 THEN
     IF B% = 1 THEN
         PRINT "Both A% and B% are 1"
     ELSE
         PRINT "A% is 1, B% is not"
     ENDIF   ELSE
         IF B% = 1 THEN
         PRINT "B% is 1, A% is not"
     ELSE
         PRINT "Neither A% nor B% is 1"
     ENDIF
ENDIF
```

The 'Dangling ELSE'

The compiler corrects one of the 'bugs' in the BASIC interpreter, by correctly translating nested single-line IF … THEN … ELSE statements. For example:

```
IF A%>B% THEN IF A%>C% THEN PRINT "A"
ELSE PRINT "C" ELSE PRINT "B or C"
```

Under the interpreter, the first ELSE applies when either of the preceding IF statements returns FALSE. This means the only possible output is "A" or "C" under all circumstances, this is clearly incorrect as demonstrated in the following table:

A	B	C	Output
1	2	3	C
1	3	2	C
2	1	3	C
2	3	1	C
3	1	2	A
3	2	1	A

Under the compiler the innermost ELSE applies to the innermost IF, and the outermost ELSE applies to the outermost IF. The correct results are returned, as follows:

A	B	C	Output
1	2	3	B or C
1	3	2	B or C
2	1	3	C
2	3	1	B or C
3	1	2	A
3	2	1	A

Multiple Exits

In addition to insisting that correct nesting of stuctures is maintained the compiler also requires that a one-to-one correspondence is maintained between all opening and closing structure markers in the program. This promotes structured programming and produces faster code. The structures are:

DEFPROC	ENDPROC	
DEFFN	=	
REPEAT	UNTIL	
WHILE	ENDWHILE	
FOR	NEXT	
CASE	ENDCASE	* These are already restricted by
IF ... THEN <RET>	ENDIF	* the BASIC V interpreter

The interpreter already applies a one-to-one rule for the CASE statement (which must have one and only one ENDCASE) and to the block structured IF (which must have a single ENDIF). For other structures however, the interpreter allows multiple exits, even when 'improper' use is made of them. This leads to ambiguity and inconsistency. For example:

```
10 DEFPROCfred
20 REPEAT
30 A = GET
40 IF A = 127 ENDPROC
50 PROCprint(A)
60 UNTIL A = 0
70 PROCchar(A)
80 ENDPROC
```

operates differently under different versions of the interpreter. Under the BASIC V interpreter, the ENDPROC on line 40 will flush the stack and discard the REPEAT UNTIL which was active. Under prior versions of the BASIC interpreter, the REPEAT UNTIL would have been left 'pending', because separate stacks were used for the different constructs.

The compiler outlaws all such operations. The above procedure should be re-written as follows:

```
10 DEFPROCfred
20 REPEAT
30 A = GET
40 IF A <> 127 THEN PROCprint(A)
50 UNTIL A = 0 OR A=127
60 IF A = 0 THEN PROCchar(A)
70 ENDPROC
```

Keyword Position

Under the interpreter there are strict rules about where certain keywords must occur in the text. These rules are more relaxed under the compiler.

The compiler still insists on the THEN of a block structured IF ... THEN ... ELSE being at the end of a line. Similarly the OF in a CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE.

However those are the only rules imposed. The keywords WHEN, OTHERWISE, ENDCASE, ENDIF etc. no longer have to be the first non-space objects on a line. For example the following is quite acceptable.

```
CASE i% OF
WHEN 1:PRINT"1":WHEN 2:PRINT"2"
OTHERWISE PRINT "ERROR":ENDCASE
```

Hints and Tips

The subject of the programming structure rules is also discussed in the *ABC User Guide*. In particular, the 'Hints and Tips' section discusses ways in which programs can be written in a manner which will be compatible with both the BASIC interpreter and ABC.

Errors and Warnings

A number of errors and warnings may results from incorrectly structured programs. These are also described in the *ABC User Guide*.

Short Cuts

The compiler does not allow the use of :

```
NEXT,
```

(NEXT comma) to terminate two (or more) FOR loops. Similarly, the use of only one NEXT statement to terminate more than one FOR loop, through the use of the outer loop's control variable only. For example, the following program is illegal:

```
10 FOR I% = 0 TO 10
20   FOR J%=0 TO 10
30   A%(I%,J%)=0
40 NEXT I%         :REM terminate both loops
```

A second shortcut which is not supported is the use of

```
UNTIL.
```

(UNTIL full stop) in place of 'UNTIL FALSE'. The compiler will not accept the dot '.' to mean 0.0.

3 : Variables and Constants

Scope Rules

It is not uncommon for a program to contain several variables with the same name. For example there may be a global name created at the 'top' level and several independent local versions, created within procedures and functions. Clearly, when a variable is referenced the program must know which one to use. This is determined by the 'scope rules'. A variable can be referenced only when it is 'in scope', so, for example a LOCAL variable can be referenced only within the procedure or function to which it belongs.

The scope rules for the compiler are 'static'. This means a specific occurrence of a non-unique variable name in the text of the program always references the same variable, whether local or global. Furthermore its meaning can be determined just by looking at the program text.

This view of the scope rules is not shared by the interpreter. Its scope rules are said to be 'dynamic'. That is, determination of which variable is referenced happens while the program is running and may depend on code elsewhere in the program. For example:

```
10 A=1
20 PROCp(A)
30 END
40 DEFPROCp(X)  : LOCAL A
50 PRINT X,A  : PROCq(X)
60 ENDPROC
70 DEFPROCq(Y)
80 PRINT Y,A
90 ENDPROC
```

Under the interpreter, the value of A in PROCq depends on where PROCq has been called from. If it is from the main program A takes the global value (1). If it is from PROCp A takes the value local to PROCp(0). Under the interpreter this program will produce:

Output		Implication
1	0	A passed to PROCp in X but A is made local to PROCp
1	0	X passed to PROCq in Y but A is still local to PROCp

The compiler classes all variables as global, unless they are declared explicitly to be local to the procedure or function in which they are used. In the example above, therefore, the local value of A applies only to PROCp, not to any other procedure. In the compiled version PROCq will always use the global value of A, regardless of where PROCq was called from. Under the compiler this program will produce:

Output		Implication
1	0	A passed to PROCp in X but A is made local to PROCp
1	1	X passed to PROCq in Y but A is now the GLOBAL value

An additional point to note about scope is that the use of GOTO or GOSUB to jump from one procedure or function into a different one will produce unpredictable results, since the variables which will be in scope after the jump are unknown.

Passing Parameters

Under both the interpreter and compiler parameters given to procedures and functions are passed by 'value'. For example:

```
...
PROCexample(A%)
...
END
...
DEFPROCexample(B%)
B% = 1
ENDPROC
...
```

In this example, A% is referred to as the *actual* parameter and B% as the *formal* parameter. When the call is made, the value of A% is copied into B%. Any subsequent alteration to B% made in the body of the procedure or function has no affect on the value of A%.

The BASIC V interpreter also supports return parameters which allow the final value of the formal parameter to be written back to the actual parameter at the end of the procedure call. This is not supported by the compiler.

Floating Point

The compiler makes use of the Floating Point Emulator (and hence the floating point co-processor if fitted). Floating point numbers are stored in IEEE standard single precision (four-bytes) rather than the non-standard five-byte method used by the BASIC interpreter. This can lead to run-time incompatibility in the following areas:

Rounding Errors

Rounding errors in floating point calculations may be different. In some cases the interpreter will be more accurate, in others the compiler will be. This should not be a problem but may mean that floating point results produced by the two may not be identical.

Floating Point Indirection

The solidus character '|' may be used to perform floating point indirection but it will transfer four bytes only. The BASIC interpreter transfers five.

EQUate

In addition to the standard EQUate statements for byte, word, double word and string constants, the compiler provides an extra assembler directive, EQUF. This allows the placing of floating point constants in the program. For example:

```
        EQUF 3.14159
```

This will store the value 3.14159 in four bytes of memory.

Floating Point in Files

The BASIC V interpreter uses a five-byte representation for floating point numbers in data files – marked by a type byte of &80. Earlier 6502 BASICs used &FF as the type marker. The compiler uses a four-byte representation and marks this with the type byte &F0. To improve compatibility, compiled programs now support both formats.

INPUT#file INPUT#file will accept either four or five byte format and does the required conversion automatically.

PRINT#file By default, PRINT#file outputs the &F0 type byte followed by the four-byte format. If compatibility is required, the &80 type-marker format can be produced by setting bit 24 of the @% print control format variable. For example: `@%=@% OR &01000000`

The format used to store floating point numbers in data files is as follows:

The type byte	Compiled programs will contain either &F0 or &80 in the type byte according to the value of @%.
The number	The four or five bytes of the floating point number will be output directly to the file, lowest byte first.

Numeric Input

Under the interpreter, the rules about what can be supplied when numeric input is called for vary depending on the keyword used. For example INPUT allows only decimal numbers, whereas READ takes decimal, hex or binary numbers, variables and some simple expressions.

Under the compiler the rules for numeric input have been standardised. Only numbers are allowed, but these may be given in decimal, hex or binary. So the format required for a number in all cases is as follows:

- An optional + or - sign
- An optional radix indicator (% or &):
  If it is % then a string of binary digits (0 or 1) follows
  If it is & then a string of hex digits (0–9 and A–F) follows
- If there is no radix indicator then a string of decimal digits (0–9)
- An optional decimal point followed by a string of decimal digits
- An optional E to introduce the exponent followed by
  an optional + or – sign, then,
  a string of up to four decimal digits

Field Format @%

Output formatting is controlled by the variable @% in both the BASIC interpreter and ABC. However, the interpretation of the four bytes of @% is slightly different for each.

Under ABC @% is interpreted as follows:

Byte four	If non-zero, then STR$ uses the print format determined by @% when converting its argument to a string. If zero, then it uses a general format. See also the section 'Floating Point in Files' above.
Byte three	Not used.

Byte two	Determines the number of decimal places for a floating point number.
Byte one	Determines the total field width.

Integers are printed in a field width given by byte one if they fit. Otherwise, they are printed in as few spaces as possible. For example:

```
@% = 5
PRINT 1
PRINT 123
PRINT 12345
PRINT 1234567
```

gives the following:

```
    1
  123
12345
1234567
```

The format of floating point numbers is determined by bytes one and two. If byte two is in the range 0 to 9, then numbers are printed in fixed point format with byte two specifying the number of places after the decimal point. For example:

```
@% = &40A
PRINT 27.234
PRINT 8
PRINT 12345.12345
```

gives approximately (allowing for floating point inaccuracy):

```
   27.2340
    8.0000
12345.1235
```

This is subject to a limit on the total number of digits of 10. If the total number of digits to be printed is greater than 10, then the number of decimal places printed is 10 minus the value of byte two. For example:

```
@% = &60F
PRINT 123456.123456
```

will give only four decimal places.

If byte two is 10 or more then general format is used. In this case numbers less than 1E-1 or greater or equal to 1E10 are printed in exponential format. For example:

```
@% = &A0A
```

```
PRINT -0.000345
```

gives a number approximately equal to:

```
-3.45E-4
```

Numbers between 1E-1 and 1E10 are printed in the form nnn.nnn. For example:

```
@% = &A0A
PRINT 27.234
```

gives approximately:

```
27.234
```

If a number does not fit in the field width given by byte one then it is printed in as few spaces as possible. The default value of @% is &A0A

4 : Running Compiled Code

Pseudo Variables

Under BASIC the pseudo-variables (eg, PAGE, TOP, etc.) hold values defining the location of your BASIC program and the workspace it uses when running. The interpreter also allows the first three of these to have values assigned to them by the user to limit the memory used.

Under the compiler, the values returned by these pseudo-variables are obviously different, since the object code is in memory during execution, not the source. Also, the compiler uses workspace differently to the interpreter. Nevertheless the pseudo-variables do return sensible values, as follows:

HIMEM	Top of memory
END	Top of heap = lower limit of stack
LOMEM	Bottom of heap
TOP	Top of the program
PAGE	Start of the program

In addition, the compiler provides an extra pseudo variable:

EXT	Current stack pointer

Figure 4.1 (opposite) shows how memory is arranged under the compiler and the location given by these pseudo variables.

The restriction in the interpreter which prevents pseudo-variable use as indirection operators does not apply to the compiler. For example:

```
PRINT END?1
```

is allowed and will work correctly. However, assignment (change) of any pseudo-variable is not permitted by the compiler.

The program's memory usage can be controlled by the commands STACK and HEAP. See Chapter Six, 'Compiler Directives' for details.

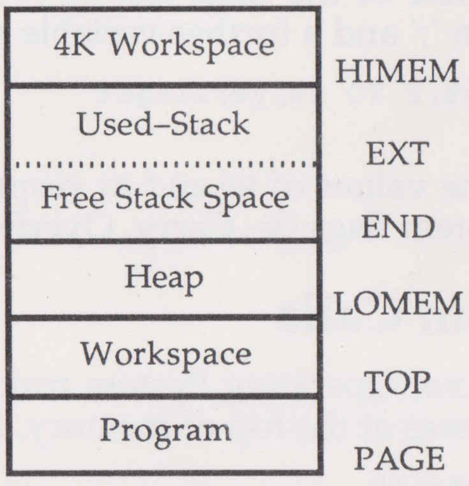


Figure 4.1. ABC memory managment.

Calling Machine Code

The two keyword for calling external machine code routines are CALL and USR. These are both available under the compiler but their syntax is different. The discrepancies are outlined below.

USR

USR takes the address of the routine to call in the usual manner. However, instead of placing the values in A% – H% in registers R0 – R7 as in BASIC programs, the USR address is followed by a list in brackets of up to eight parameters which will be assigned to the registers. It is not necessary to supply all eight values, but those supplied will be assigned in sequence, and any register whose contents are not specified should be regarded as unknown. The single value returned by the USR function is the value held in R0 when the routine ends. For example:

```
PRINT USR (routine%,1,5,0)
```

CALL

CALL is similar to USR but has been extended so that it can be followed by the keyword TO and a list of up to eight variable names, into which the values in R0 – R7 are placed when the routine ends. For example:

```
CALL code%,2 TO x%,y%
```

This provides a method of getting several values back from a routine at once. In addition the state of the flags can be returned by following the variables by a semicolon ';' and a further variable name. For example:

```
CALL code%,2 TO x%,y%;flags%
```

x% and y% will hold the values of R0 and R1 respectively and flags% will hold the state of the system flags (ie, Carry, Overflow, Zero etc.).

Operating System Calls

In a 6502 based BBC micro, Operating System routines can be accessed by calling particular addresses at the top of memory, for example:

```
10 A%=138
20 X%=0
30 Y%=65
40 CALL &FFF4
```

This program calls the OSBYTE routine in the operating system to insert an "A" into the keyboard buffer.

The BASIC interpreter on the Archimedes has a list of 'legal' 6502 entry points and checks every CALL or USR to see if the address called is one of these. If it is the interpreter translates the call into the equivalent SWI.

This attempt at compatibility with earlier Acorn micros has a problem. These addresses now lie in user memory because the very large address space and increased amount of RAM has meant that the ROM is at &03800000 upwards. Hence, if you generate object code and try to CALL it you may be unlucky and hit one of these special addresses. If this occurs the 6502 routine will be called, not your own code.

For this reason, and the fact that the compiler is aimed at helping the development of new code rather than running existing code, this feature is not supported by the compiler. If the compiler observes a call of this nature it will give a warning.

Array Handling

The compiler supports all of the array handling features of BASIC IV. It provides neither the extended features of BASIC V for handling whole arrays nor the LOCAL array handling. One further restriction is that array elements may not be used as the control variables of FOR loops.

Items affected in this class are:

The DIM function
The SUM functions
Passing arrays as parameters
Array assignment
Array arithmetic and vector processing
LOCAL arrays

Error Handling

The compiler supports all of the error handling features of BASIC version IV with the exception that the ERL function cannot be used to determine the line number on which an error occurs.

LOCAL Errors

There is no facility for LOCAL error handling. All errors will return the stack to the top level and thus all LOCAL variables will be lost. This can result is the wastage of memory if an error occurs in a procedure which has LOCAL string variables, string parameters or which requires some temporary string workspace.

After an error the global variables will be 'in scope' and so any error handler should not cause a GOTO into the scope of a procedure. It is perfectly correct to perform

```
ON ERROR PROCerror:GOTO 1000
```

provided that line 1000 forms part of the main program.

It is worth noting that the scope rules for the interpreter are rather strange in this respect. After an error, any variable name for which both an active LOCAL and a global existed at the time the error occurred will retain the value which was held in the local variable. This could potentially cause havoc and is not the case under the compiler.

Miscellaneous Differences

TAB(n) and SPC(n) do not use COUNT (which isn't implemented). TAB(n) outputs spaces until POS = n. SPC(n) outputs n spaces.

Commands

The following BASIC keywords are commands that cannot be used in programs either under the BASIC interpreter or the compiler:

AUTO	LVAR
DELETE	NEW
EDIT	OLD
HELP	RENUMBER
INSTALL	TWIN
LIBRARY	TWINO
LIST	TRACE

If the compiler discovers one in a source program it will terminate compilation, with the message:

```
Error :This operation is not supported
30 TRACE ON
```

Program Manipulation

The compiler forbids the use of commands which load or store BASIC programs since at run time the source program is not available. The use of any of the following keywords in a compiler source program will cause an error to be reported during compilation:

APPEND
CHAIN
CLEAR
LOAD
SAVE

Other Keywords

The following list of miscellaneous keywords are not supported by the compiler and will cause an error if used:

COUNT
EVAL
SUM
WIDTH

The second of these, EVAL, would require that the compiled program to have access to the complete source-text of the program during execution and the means to decode it. This would be analogous to including a copy of the interpreter with every program and is therfore not a viable option!

COUNT and WIDTH are not implemented since they are so rarely used that the reduction in the speed of output routines which they cause is not justified.

SUM is not supported since it forms part of the whole-array manipluation package which is not available.

Indirection Assignments

Whilst reading a value from an indirect expression is fully supported, assignment to them is restricted to use in straightforward assignment statements. The following are allowed:

```
PRINT $(buffer%+I%)
PROCfred(A%?B%)
$(buffer%+I%)="Hello"
```

But the following types of operation are *not* allowed:

```
INPUT $buffer%
INPUT #file%,buf%?I%
SWAP I%!(J%+o),I%!(J%+4)
MOUSE B%!0,B%!4,B%!8
[:.lables%!I% :]
READ $buffer%
DEF PROCaction(block%?1)
LOCAL !FNfred
SYS A,B,C to $buffer%
LEFT$(buf%)="XXX"
```

DATA and READ

Under the BASIC interpreter it is possible to place expressions such as 10*20 in DATA statements and have them evaluated when READ. This also applies to variable names. For example:

```
READ A%,B%
DATA X*StepX, 100
```

would be legal and result in A% being set to the value of X*StepX and B% to 100. (This assumes that X and StepX both exist.)

The compiler cannot support this in either case since it would require the expression evaluator EVAL. Unfortunately, it is not possible for the compiler to detect this situation since the data could be used quite legally as a string. Therefore a program such as this will compile but probably not run as expected. The example above will result in A%=0 and B%=100.

5 : The In-line Assembler

The in-line assembler is fully supported. The facility is included for compatibility, where this is vital. Note, however, that it is far more efficient and logical to assemble machine code as separate external routines – then call these as general subroutines – than to include them in a compiled program.

Remember that the assembler does not take over from the compiler, you will actually be compiling a program which assembles a program at run-time! If the facility is used, however, certain differences should be noted.

Registers

The register names must be explicitly defined. In the interpreter it is permitted to call the registers R0 – R15 by default. These variables can be assumed to automatically contain the values 0 – 15, and hence refer to the registers. Under the compiler it is always necessary to assign the register's names before use.

Two-Pass Assembly

Two passes are still required to produce code with any forward references resolved. However, it is interesting to note that undefined label references will be found by the compiler rather than by the assembler. The meaning of bit one in the OPT command (enable/disable assembler errors) is therefore altered.

Assembly Listings

Bit zero of the OPT setting (enable/disable assembly listing) is available, however the compiled code cannot produce a full listing because to do so would require access to the source text. For instance branches are shown as the mnemonic followed by the address of the destination. All information about the name of the label having been lost.

EQUate Directives

In addition to the standard EQUate directives for byte, word and string, the EQUF directive has been added to support floating point constants. This places a single-precision floating point operand in memory, occupying four bytes.

In General

BASIC treats anything within square brackets, '[' and ']', as assembly language. These can appear anywhere in a BASIC program. However, before including assembly language statements there are certain house keeping chores to undertake. First, you must reserve memory for the machine code to assemble into. This can be done using DIM. For example:

```
DIM code% 100
```

This reserves 101 bytes of memory starting at the address code%. You must then point to this area so the assembler places the machine code there. This is done by setting P%, the 'program counter' to point the start of the block, for example:

```
P% = code%
```

The first assembler instruction will then be placed at the address in P% and the value will be incremented so that it points to the next free location, ready for the next instruction, and so on.

An alternative is 'offset assembly'. This causes machine code to be assembled as if it is to run at one address, but it is actually assembled into another. To do this, O% is set to point to the address where the instructions will assemble. P%, as before, points to the address where the code will execute. For example:

```
O% = code%
P% = 1000
```

To notify the Assembler that this method of generating code is to be used, the directive, OPT, has bit two set, eg, OPT 4.

Two pass assembly is often required to generate machine code. The assembler reads the assembly language instructions twice before generating the machine code program. This is needed whenever an instruction causes execution to 'jump' forwards in the program, to a label which the assembler hasn't yet encountered, and therefore doesn't know the address for. This is the normal situation, and the two passes are performed by a FOR ... NEXT loop as follows:

```
DIM code% 400
FOR pass% = 0 TO 3 STEP 3
P% = code%
[
OPT 0
...
]
NEXT pass%
```

Note that the pointer(s), in this case just P%, must be set at the start of both passes. If O% were used for offset assembly it too would need to be set each time, so the loop might be:

```
DIM code% 400
FOR pass% = 4 to 7 STEP 3
P% = 1000
O% = code%
[
... etc
```

Another task is to assign values to any variables which will be used within the assembly language. In particular this is necessary for the 'register' names. All ARM instructions act on registers, which are special memory locations in the processor. Sixteen are available at any one time. They can be referred to in the text as 0 – 15 but it makes the text easier to read if variables such as R0 – R15 are explicitly assigned instead. R15 is often refered to as PC, eg, assembly listings from OPT3 will always give R0 – R14 and PC, whatever they were called in the source text.

Assembler Instructions

This section describes the different categories of instructions and assembly directives available. Their syntax is described in the following terms:

Abbreviation	Meaning
< >	Indicates that the contents of the brackets are optional
(x | y)	Indicates that either x or y, but not both, may be given
#exp	Indicates that an expression is to be used which evaluates immediately to a constant
Rn	Indicates that an expression evaluating to a register number should be used, eg, just a register name shift indicates that one of the following shift options should be used:

Opcode		Action
ASL	(Rn | #exp)	Arithmetic shift left by contents of Rn or immediate expression
LSL	(Rn | #exp)	Logical shift left
ASR	(Rn | #exp)	Arithmetic shift right
LSR	(Rn | #exp)	Logical shift right
ROR	(Rn | #exp)	Rotate right
RRX		Rotate right one bit with extend

One further point to note applies to all ARM instructions. They can be performed conditionally, based on the status of one or more of the following flags:

Flag	Purpose
C	Carry flag
N	Negative flag
V	Overflow flag
Z	Zero flag

The sixteen condition codes available are:

Code	Meaning	Flags
AL	Always	
CC	Carry clear	C=0
CS	Carry set	C=1
EQ	Equal	Z=1
GE	Greater than or equal	N=1 AND V=1 or N=0 AND V=0
GT	Greater than	N=1 AND V=1 AND Z=0 or N=0 AND V=0 AND Z=0
HI	Higher (unsigned)	C=1 AND Z=0
LE	Less than or equal	N=1 AND V=0 or N=0 AND V=1 or Z=1
LS	Lower or same (unsigned)	C=0 or Z=1
LT	Less than	N=1 AND V=0 or N=0 AND V=1

MI	Negative	N=1
NE	Not equal	Z=0
NV	Never	
PL	Positive	N=0
VC	Overflow clear	V=0
VS	Overflow set	V=1

Two of these may be given alternative names as follows:

LO	Lower unsigned is equivalent to CC
HS	Higher/same unsigned is equivalent to CS

Arithmetic and Logical Instructions

The instructions available are:

Instruction	Result of opcode	Rd, Rn, Rm
ADC	Add with carry	Rd = Rn + Rm + C
ADD	Add without carry	Rd = Rn + Rm
SBC	Subtract with carry	Rd = Rn – Rm – (1 – C)
SUB	Subtract without carry	Rd = Rn – Rm
RSC	Reverse subtract with carry	Rd = Rm – Rn – (1 – C)
RSB	Reverse subtract without carry	Rd = Rm – Rn
AND	Bitwise AND	Rd = Rn AND Rm
BIC	Bitwise AND NOT	Rd = Rn AND NOT (Rm)
ORR	Bitwise OR	Rd = Rn OR Rm
EOR	Bitwise EOR	Rd = Rn EOR Rm

Instruction	Result of opcode	Rd, Rm
MOV	Move	Rd = Rm
MVN	Move NOT	Rd = NOT Rm

The general format for these instructions is:

```
opcode<cond><S>Rd,<Rn>,(#exp I I Rm <,shift>)
```

Each of these instructions produces a result which it places in a destination register (Rd). The instructions do not affect bytes in memory directly. As noted above, all of these instructions can be performed conditionally. In addition, if the 'S' is present, they can cause the condition codes to be set or cleared, for example:

```
ADDEQ     R1, R1, #7
SBCS      R2, R3, R4
AND       R3, R1, R2, LSR #2
```

Special actions are taken if any of the registers is R15, the program counter. If one of the source registers is R15 then:

- If Rm = R15 all 32 bits of R15 are used in the operation, ie, the PC + flags
- If Rn = R15 only the 24 bits of the PC are used in the operation

If the destination register is R15 then the action depends on whether the optional 'S' has been used:

- If S is not present only the 24 bits of the PC are set
- If S is present the whole result is written to R15

Multiply Instructions

There are two instructions associated with multiplication:

Instruction	**Operation**	**Rd, Rm, Rs, Rn**
MUL	Multiply	Rd = Rm * Rs
MLA	Multiply-accumulate	Rd = Rm * Rs + Rn

Syntax:

```
MUL<cond><S> Rd,Rm,Rs
MLA<cond><S> Rd,Rm,Rs,Rn
```

Examples:

```
MUL R1,R2,R3
MLAEQS R1,R2,R3,R4
```

The multiply instructions perform integer multiplication, giving the least significant 32 bits of the product of two 32-bit operands.

The destination register must not be R15 or the same as Rm. Any other register combinations can be used.

If the 'S' is given in the instruction, the N and Z flags are set on the result, the V flag is unaffected and the C flag is undefined.

Comparisons

There are four comparison instructions:

Instruction	Operation	Rn, Rm
CMN	Compare	Rn + Rm
CMP	Compare	Rn – Rm
TEQ	Test equal	Rn EOR Rm
TST	Test	Rn AND Rm

Syntax:

opcode<cond><P> Rn, (#exp | Rm <,shift>)

Example:

```
CMP R0,R1
```

These are similar to the arithmetic and logical instructions except that they do not have a destination register since they do not return a result. They also automatically set the condition flags (since they would perform no useful purpose if they didn't). Hence, the 'S' of the arithmetic instructions is implied.

There is an additional function of these routines which is to set the whole of the PSR to a given value. This is done by using a 'P' after the opcode, eg CMNP. Normally the flags are set depending on the value of the comparison. The 'P' option allows the corresponding bits of the result of the calculation performed by the comparison to overwrite the flags.

Branching

There are essentially only two branch instructions but in each case the branch can take place as as a result of any of the 16 condition codes:

Instruction	Result
B(cond)	Branch
BL(cond)	Branch and link

Syntax:

opcode<cond> <exp>

Examples:

```
B lab1

BLEQ equal
........
........
.equal
........
MOV R15 R14
```

The branch instruction causes the program execution to jump to the instruction given at the specified address. This address is held relative to the current location, and is defined by a named label.

The branch and link instruction performs the additional actions of copying the address of the instruction following the branch into register R14 and storing the condition codes there as well. R14 is known as the 'link register'. This means that the routine branched to can be returned from by transferring the contents of R14 into the program counter and can restore the flags from this register on return. Hence instead of being a simple branch the instruction acts like a subroutine call.

Single Register Load/Save

The single register load/save instructions are as follows:

Instruction	**Result**
LDR	Load register
STR	Store register

Syntax:

opcode<cond><B><T> Rd, address

These instructions allow a single register to load a value from memory or save (store) a value to memory at a given address. Addresses are always enclosed in square brackets. The simplest form an address can take is a register number in which case the contents of the register are used as the address to load from or save to.

Another option is to use the contents of a register after these contents have had other values added or have been shifted. The register can be optionally updated to contain the address which was actually used by adding a '!' after the closing square bracket. This is known as pre-indexed addressing since the address written to the register is calculated before the load/save takes place.

Syntax	Address
[Rn]	Contents of Rn
[Rn,#m]	Contents of Rn + m
[Rn,Rm]	Contents of Rn + contents of Rm
[Rn,Rm,shift #s]	Contents of Rn + (contents of Rm shifted by s places)

The alternative is post-indexed addressing. In all cases the address being used is given solely by the contents of the register Rn. The rest of the instruction determines what value is written back into Rn. This write back is performed automatically; no '!' is needed. It gets its name from the fact that the address written to the register is calculated after the load/save takes place.

Syntax	Value written back
[Rn],#m	Contents of Rn + m
[Rn],Rm	Contents of Rn + contents of Rm
[Rn],Rm,shift #s	Contents of Rn + (contents of Rm shifted by s places)

Constants in Object Code

The assembler provides the following directives:

EQUB	Reserve one byte of memory
EQUW	Reserve two bytes of memory
EQUD	Reserve four bytes of memory for an integer value
EQUF	Reserve four bytes of memory for a fp number
EQUS	Reserve memory for a string

These operations initialise the reserved memory to the values specified by the address field. EQUB, EQUW and EQUS should be followed by an ALIGN which aligns the program counter to a word boundary. All instructions must occur on word boundaries so failing to use ALIGN will cause a 'Bad alignment' message to occur, when the next instruction is encountered. EQUD and EQUF both allocate four bytes, so no ALIGN is required after them.

```
EQUF 35.5
EQUS "Hello" ALIGN
```

Multiple Register Load/Save

These instructions allow the loading or saving of several registers:

Instruction	Result
LDM	Load multiple registers
STM	Store multiple registers

Syntax:

opcode<cond>(I | | D)(A | | B) Rn<!>,{Rlist}<^>

Examples:

```
LDMIA R5, {R0,R1,R2}
LDMDB R5, {R0,R1,R2}
```

The contents of register Rn give the base address from/to which the value(s) are to be loaded/saved. Rlist provides a list of registers which are to be loaded from/saved to. The order the registers are given in the list is irrelevant since the lowest numbered register will be loaded first and the highest one last. For example, a list comprising {R5,R3,R1,R8} will be loaded from/saved to in the order R1, R3, R5, R8.

The 'I' or 'D' indicates whether the addresses being loaded from/saved to are to increase or decrease from the base address. Hence it is possible to load/save a series of registers from/to a base address and the locations either above it or below it in memory.

The 'A' or 'B' indicates whether or not the addresses are to be altered after or before each register is loaded/saved. If they are being altered afterwards, then the first register will be loaded from/saved to the base address, then the address will be either increased or decreased before the next load/save. If they are to be altered before, then the base address will be either increased or decreased before the first load or save takes place, then again before the second, etc.

There are two further options. If a '!' is present after the register containing the base address then this register will be updated to contain the final address.

If a '^' is given at the end of the register list on a load and R15 is contained in this list, then the whole 32 bits of R15 will be loaded.

Operating System Functions

Syntax:

```
SWI <expression>
```

The SWI mnemonic stands for SoftWare Interrupt. A SWI call enables all the Operating System routines to be accessed directly. The expression contains the identifier of the SWI only. Any parameters required must be set up in the appropriate registers immediately before the call is issued.

6 : Compiler Directives

To control the actions of the compiler a number of 'compiler directives' are available. These take the form of REM statements and as such are ignored if the program is RUN under the interpreter. See also Chapter 20 for compiler directives associated with compilation of relocatable modules.

Listing the Source Code

A pair of directives exists to to switch source code listing on or off as the program is compiled. By default the source program will be listed out in full on each pass. You may suppress this by placing:

```
REM {NOLIST}
```

in the program at the point where you want to disable listing.

To re-enable listing use:

```
REM {LIST}
```

Turning off the listing reduces the time taken for compilation.

Ignoring Sections

If there is a section of a program which you do not wish to be compiled, for instance debugging procedures, you can instruct the compiler to omit it. This is accomplished by placing the directive:

```
REM {NOCOMPILE}
```

before the section to be ignored and by placing:

```
REM {COMPILE}
```

following it to resume compilation.

This provides a much better solution than 'commenting-out' a whole section. Note that the compiler will flag an error if you attempt to nest these directives.

For existing code, most of the differences listed above can be dealt with by making minor modifications. However, there may be some circumstances in which the code required for the compiler is different to that for the interpreter, for example when dealing with CALL and USR.

To avoid the need for two versions of a program, one for the interpreter, one for compilation, the following 'trick' can be used. Write two procedures, with the same name, one containing the code needed for the interpreter and one for the compiler. Place these at the end of the program, the interpreter one first, and surround the interpreter procedure by the 'NOCOMPILE' and 'COMPILE' directives. For example:

```
MAIN PROGRAM
......
PROCdiffer
......
END
:
REM {NOCOMPILE}
DEFPROCdiffer
CALL code%
ENDPROC
REM {COMPILE}
:
DEFPROCdiffer
CALL code%,A%,B%
ENDPROC
```

When the program is run under the interpreter, the first version of the procedure will be found and used. The compiler directives are simply treated as remarks, and so are ignored while the fact that this procedure name exists twice does not cause an error. This is because the interpreter always searches from the beginning of the program for procedures. When the name is found the search terminates.

However, the REM statements instruct the compiler to ignore the interpreter version of this procedure, so only the second version will be included in the compiled program.

Use of Memory

There are two directives which control the way in which your program will allocate memory to its various tasks. If neither of these is specified then a sensible default setting will be used.

When a compiled program is running it requires three separate areas of workspace, the BASIC Heap, the BASIC Stack and the System Area, the functions of each of these are outlined below.

The BASIC Heap

This is where all global variables, strings and arrays are held. It is also where any DIM statement will claim memory from. It is located immediately above the top of the program in memory.

The BASIC Stack

This is where all parameters, local variables and procedure call information in held. It is located high up in memory and grows downwards towards the top of the heap.

The System Area

This is a 4k area for a small stack and some private information. This will be at the highest address in memory which is used.

The default action is to reserve 4K at the top of memory for the system area and then divide the remainder equally between the heap and the stack. In most cases this will be suitable, but some programs will need more of one type of memory than of the other. For example a program which operates on very large arrays may well need most of memory to hold them whilst at the same time having only very modest stack space requirements.

To give you control over this, the amount of heap and stack space required by a program can be set explicitly. For example, if you know in advance that your program needs 10k of heap-space you could include the directive:

```
REM {HEAP = 10000}
```

It is difficult to calculate the exact amount of space that a program will use so it is a good idea to ask for a bit too much!

Alternatively the stack size can be specified as follows:

```
REM {STACK = 6000}
```

This will reserve 6000 bytes of memory for the stack. When your program is running, HIMEM will return the address of the top of this stack and the EXT function will return the current value of the stack pointer. Thus the amount of stack space in use at any given time is:

```
HIMEM-EXT
```

If you specify either HEAP or STACK, but not both, then the other will use up the rest of the available memory. Thus, a good way to maximise the amount of heap space is to specify the minimum stack size and leave the heap to grab the rest of memory.

If both options are specified then the two areas will be adjacent in memory as shown in figure 6.1. There will be a free area of memory above the top of the system area as indicated.

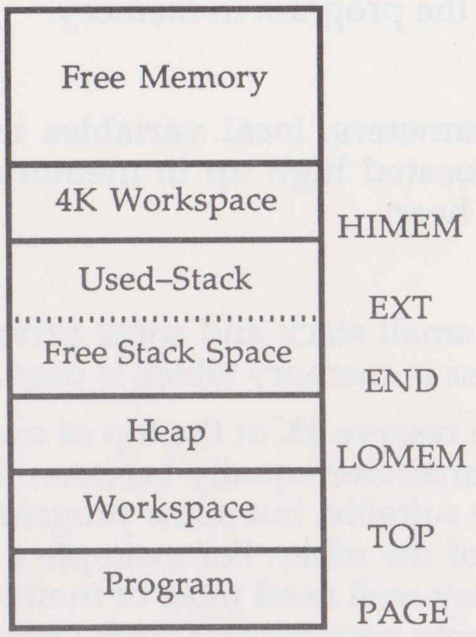


Figure 6.1. Use of memory.

Stack Limit Checking

The stack, as stated above, grows downwards towards the top of the heap. If the two collide the error:

```
stack overflow
```

is generated. However, the overhead in checking the stack pointer on every procedure call can be quite significant and so an option is provided to prevent the compiler generating the code to do so.

To force the compiler to compile procedures and functions without stack checking code from that point in the program onwards use:

```
REM {NOSTACKCHECK}
```

To re-enable stack limit checking code generation use:

```
REM {STACKCHECK}
```

Checking for ESCAPE

By default, the compiler will produce machine code which will respond to the ESCAPE key being pressed. This either calls the program's own error handler, (if it contains one), or by giving the message 'ESCAPE' and terminating execution.

To provide the default ESCAPE action the compiler must include special code in the object program at regular intervals. This makes the compiled code both larger and slower. An option is provided to turn ESCAPE checking off:

```
REM {NOESCAPECHECK}
```

This causes the compiler to omit the ESCAPE checking code from that point in the program onwards. When this part of the code is executed, pressing ESCAPE will have no effect. This should be used when you have catered for the ESCAPE key in your own error handler, or when you do not wish to permit the ESCAPE key to have any effect. This does not however disable the normal action of ESCAPE with respect to INPUT operations such as GET. These will still respond in the usual way.

To re-enable ESCAPE checking code generation use:

```
REM {ESCAPECHECK}
```

Assembler Directives

There are two other compiler directives which can be used reduce the size of the object code. To ignore and re-enable the OPT directives respectively use one of the following:

```
REM {NOOPT}
REM {OPT}
```

Whenever the assembly is re-started the compiler will use the OPT value last used, for example:

```
10 FOR opt%=0 TO 3 STEP 3
20 [OPT opt%:]
30 PRINT "About to restart assembly"
40 REM {NOOPT}
50 [.freed:]
60 [.jim:]
70 NEXT
```

Lines 50 and 60 will use the OPT value set at line 20. As you can see the 'OPT opt%' has been left out when assembly has been re-started. Unfortunately this will render your source code incompatible with the interpreter.

7 : Structures

This chapter describes all the program flow control structures which are available. These form the heart of any BASIC program. They fall into the following three categories:

1. Conditional Structures

These provide a branch mechanism within programs, causing different statements to be executed under different circumstances.

2. Loop Structures

These allow a series of statements to be executed repeatedly, either for a fixed number of times or until a certain condition holds.

3. Procedures and Functions

These allow series of statements forming a particular logical task to be grouped together and coded once. A procedure or function can then be executed from anywhere within the program. They enable large programs to be split into smaller, easy to manage sections.

Conditional Structures

The followng structures are covered here:

```
CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE
PROC + DEFPROC ... ENDPROC
FN + DEFFN ... =
```

CASE ... OF ... WHEN ... OTHERWISE ... ENDCASE

Allows different actions to be made depending on the result of an expression

Syntax:

```
CASE <expression> OF
WHEN <expression> {<,expression>}<statements>
WHEN <expression> {<,expression>}<statements>

.........

[OTHERWISE] <statements>
ENDCASE
```

Example:

```
CASE a$ OF
WHEN "A", "B", "C" : PRINT "START"
WHEN "X", "Y", "Z" : PRINT "END"
OTHERWISE PRINT "MIDDLE"
ENDCASE
```

CASE is followed by any numeric or string expression. The result of this expression is checked against the values of each of the expressions in the list following the first WHEN statement. If there is no match for any WHEN, the next WHEN in sequence is tested, (if there is one). If a match is found, the block of statements following the matched WHEN, down to the next WHEN or OTHERWISE or ENDCASE will be executed.

Only the statements of the first matched WHEN are executed, any subsequent ones are ignored. After a match control passes to the next statement following the ENDCASE.

The OTHERWISE statement is executed only if all the preceding WHEN statements have failed to match the value of the CASE expression. OTHERWISE matches any values. Control then jumps to the statement following the ENDCASE.

The OF statement must be the last item on the line. WHEN, OTHERWISE and ENDCASE can occur anywhere on a line.

PROC + DEFPROC ... ENDPROC

Defines and calls procedures.

Syntax:

```
PROC<name>[<expressions>]
DEFPROC<name> [(<variable>,...<variable>)]
.....
ENDPROC
```

Example:

```
PROCinfo("Peter",4)
PROCinfo(boy$,a%)
END
DEFPROCinfo(name$,age%)
PRINT name$;" is ";age%;" years old"
ENDPROC
```

The DEFPROC statement must be the first non-space object on a line. It marks the first line of a user defined procedure. Each DEFPROC must have one and only one corresponding ENDPROC marking the end of the procedure code.

The optional list of variables following the definition are the parameters to the procedure. They can be integers, floating point numbers or strings but *not* whole arrays. The parameters are local to the procedure, and are used within it in place of the actual parameters supplied when it is called. When the procedure is called it must be supplied with the correct numbers and types of parameter. The interpreter allows the formal parameter to be an indirect expression, which is not allowed by the compiler. For example:

```
DEFPROCfred(?&DEAD)
...
ENDPROC
```

The formal parameters must be simple variables (see also LOCAL).

Procedure definitions should be placed at the end of the program, after the terminating 'END' statement, or in such a way that they cannot be executed except when called by an appropriate PROC statement.

FN + DEFFN ... =

Defines and calls functions.

Syntax:

```
<result>=FN<name>[(<exp>,...,<exp>)]
DEFFN<name>[(<variable>,...<variable>)]

.....
=<exp>
```

Example:

```
av = FNaverage(25.5,27.3,28.8)
END
DEFFNaverage(val1,val2,val3)
= (val1 + val2 + val3) / 3
```

The DEFFN statement must be the first non space object on a line. It marks the first line of a user defined function. Each DEFFN must have one and only one statement beginning with an equals sign '='. This assigns a single result to the function and marks the end of the body of the function code. The result can be of any data type.

The optional list of variables following the definition are the parameters to the function. They can be integers, floating point numbers or strings but not whole arrays. The parameters are local to the function, and are used within it in place of the actual parameters supplied when it is called. When the function is called it must be supplied with the correct numbers and types of parameter.

Function definitions should be placed at the end of the program, after the terminating 'END' statement, or in such a way that they cannot be executed except when called by an appropriate FN statement.

FOR ... TO ... STEP ... NEXT

Allows one or more statements to be executed a specified number of times.

Syntax:

```
FOR <variable>=<exp> TO <exp> [STEP<exp>]
.....
.....
NEXT [<variable>]
```

Examples:

```
FOR I=1 TO LEN(a$)
PRINT MID$(a$,I,1)
NEXT

FOR X = 5 TO -5 STEP -1
FOR Y = 5 TO -5 STEP -1
CIRCLE X*100, Y*100, 50
NEXT Y
NEXT X
```

The FOR is followed by the 'loop variable' which must be numeric. This variable is initially assigned the value of the first expression, before the statements down to the NEXT are executed. At the NEXT statement, the loop variable is amended by the step size (or plus one if this is omitted) and compared with the value of the TO expression. If the required TO value is not met, the loop repeats until it is. Note that because the TO value is tested after the loop has executed, these statements always execute at least once, regardless of the initial value of the loop variable.

The value of the loop variable when the loop has finished should be treated as 'undefined', and should not be used again before being reset by an explicit or implicit assignment.

The loop variable can be given after the NEXT for clarity but is not required.

GOSUB ... RETURN

Calls a subroutine.

Syntax:

```
GOSUB <factor>
......
RETURN
```

Example:

```
GOSUB 200
```

GOSUB is followed by an integer factor which must yield a value between zero and 65279, ie a valid BASIC line number. The line number given will be jumped to, and control will be returned to the statement after the GOSUB by the first RETURN statement encountered.

GOTO

Transfers control to another line.

Syntax:

```
GOTO<factor>
```

Example:

```
GOTO 1000
GOTO (A%+3)
```

GOTO is followed by an integer factor which must yield a value between zero and 65279, ie, a valid BASIC line number. The line number is jumped to and execution carries on from this new line. The line must exist or an error will be given either at compile time when the line number can't be found, or at run time when a computed line can't be found.

IF ... THEN ... ELSE ... ENDIF

Allows conditional execution of one or more statements, depending on the result of a conditional expression. It comes in two forms:

First Form

Syntax:

IF <exp>[THEN][<statements>][ELSE][<statements>]]

Examples:

```
IF flag% THEN 220
IF a% > b% THEN c% = a% ELSE a% = -a% : b% = -b%
```

IF is followed by a conditional expression. If this expression yields a non-zero value it is counted as TRUE, otherwise it is counted as FALSE. If TRUE the statements in the THEN part are executed, otherwise the statements in the ELSE part are executed.

The THEN keyword is optional except in certain circumstances. These include an assignment to a pseudo-variable, before a * command and before a line number when a GOTO is implied.

Second Form

Syntax:

IF <exp> THEN <statements> ELSE<statements> ENDIF

Examples:

```
IF A > 0 THEN
        PRINT "Square root of A = ";SQR(A)
ELSE
        PRINT "No square root"
ENDIF
```

IF is followed by a conditional expression. If this expression yields a non-zero value it is classed as TRUE, otherwise it is FALSE. If TRUE the statements on the line following THEN down to either an ELSE or an ENDIF are executed.

If FALSE the statements between the ELSE until the ENDIF are executed. The THEN statement must be the last item on the line. The ELSE and ENDIF statements may occur anywhere on the line.

ON ... GOSUB ... ELSE

Provides a means of calling different subroutines depending on the value on an integer expression.

Syntax:

```
ON<exp>GOSUB<factor>[,<factor>][ELSE<statement>]
```

Example:

```
ON age% GOSUB year%+1,year%+2,year%+3 ELSE GOSUB age%
```

ON is followed by an integer expression, then the keyword GOTO and a list of factors which are treated as line numbers. If the integer expression produces the value n, then the subroutine starting at the nth line number in the list is called.

If the expression gives a value less than one or a value greater than the number of line numbers in the list then either the single statement following the ELSE is executed (if the ELSE statement exists) or an error is generated if no ELSE statement is provided.

ON ... GOTO ... ELSE

Provides a means of branching to different lines depending on the value on an integer expression.

Syntax:

```
ON <exp>GOTO<factor>[,<factor>][ELSE<statement>]
```

Example:

```
ON X% GOTO 100,200,300 ELSE PROCerror
```

ON is followed by an integer expression, then the keyword GOTO and a list of factors which are treated as line numbers. If the integer expression produces the value n, then the nth line number in the list is jumped to.

If the expression gives a value less than one or a value greater than the number of line numbers in the list then either the single statement following the ELSE is executed (if the ELSE statement exists) or an error is generated if no ELSE statement is provided.

ON ... PROC ... ELSE

Provides a means of calling different procedures depending on the value on an integer expression.

Syntax:

```
ON <exp> <PROC>[,<PROC>][ELSE<statement>]
```

Example:

```
ON a%+b%+1 PROCneg1(a%+b%),PROCzero,PROCpos1(a%+b%)
```

The ON is followed by an integer expression and then a list of procedure calls. If the integer expression produces the value n, then the nth procedure in the list is called.

If the expression gives a value less than one or a value greater than the number of line numbers in the list then either the single statement following the ELSE is executed (if the ELSE statement exists) or an error is generated if no ELSE statement is provided.

REPEAT ... UNTIL

Allows repetition of one or more statements until a certain condition occurs.

Syntax:

```
REPEAT
    [<statements>]
UNTIL <condition>
```

Examples:

```
REPEAT UNTIL NOT INKEY-99
REPEAT   X% = X% + 1 : Y% = Y - 1 : UNTIL Y% < 0
```

The statements following REPEAT will be executed until the condition following the matching UNTIL evaluates to FALSE. The statements will always be executed at least once, regardless of entry conditions. The statements may occur over several program lines, or may all be on the same line separated by colons.

WHILE ... ENDWHILE

Allows repetition of one or more statements while a certain condition prevails.

Syntax:

```
WHILE <conditional expression>
....
ENDWHILE
```

Example:

```
WHILE TIME < 1000
CIRCLE RND(1000),RND(1000),RND(200)
ENDWHILE
```

WHILE is followed by an expression. If this yields TRUE (non-zero) the statements down to the ENDWHILE are executed and then the expression is evaluated again. If it yields FALSE (zero) the statements down to the ENDWHILE are not executed and control passes forward to the statement immediately after the corresponding ENDWHILE.

If the initial value of the expression is FALSE the statements making up the WHILE ... ENDWHILE loop are never executed.

8 : Arithmetic Instructions

This chapter contains details of all the arithmetic instructions. These includes operations on integers, for example, integer division and remainder, bitwise operations such as AND and OR and arithmetic functions such as square root, sine, cosine and logarithms.

In case you are unfamiliar with the terminology, floating point numbers, also called real numbers, are numbers which may have a fractional part eg, -2.714, 965.5. Integer numbers have no fractional part, eg, 2715, -1. Computers deal with each type in a completely different way. A number may still be real, but equal to an integer, eg, 24.0

Floating point operations are performed using the Floating Point Emulator or the floating point co-processor if fitted. Calculation is performed to single-precision.

In general integer and floating point numbers are interchangeable when given as arguments to keywords. For example if the integer 5 is given when a floating point number is expected it will be treated as 5.0, and used normally. Conversely if 5.1 is given when an integer is expected, INT(5.1), ie, 5 will be used. This also applies when the result of an expression is assigned to a variable. For example:

```
number = 5/2
```

assigns the floating point value 2.5 to number whereas

```
number% = 5/2
```

assigns the value 2 to number%.

Details of syntax with examples of each of the arithmetic instructions and functions along with a brief overview can be found on the following pages. Commands are arranged in alphabetical order.

ABS

Returns magnitude of a numeric value.

Syntax: <result> = ABS <factor>

Examples:

```
posroot = ABS root
length = ABS(vec(1)^2 + vec(2)^2)
```

ABS returns the same value as its argument if this is positive, or – (the argument) if it is negative. Note that the largest negative integer does not have a legal positive value, so that if a%=–2147483648, ABS a% yields the same value, –2147483648.

ACS

Returns the arc-cosine of a numeric value.

Syntax: <result> = ACS <factor>

Examples:

```
ang = ACS FNinputang
ang1 = ACS(nv1(1)*nv2(1) + nv1(2)*nv2(2))
```

ACS must be followed by a factor yielding a value between –1 and +1 inclusive. It returns the arc-cosine of this value in radians as a real, in the range 0 to PI.

AND

Returns the logical or bitwise AND of two integer values.

Syntax: <result> = <factor> AND <factor>

Examples:

```
IF flag1 AND flag2 THEN PRINT "OK"
PRINT x% AND y%
```

AND must be both preceded and followed by a factor yielding an integer value. It returns the value resulting from ANDing the corresponding bits in these two values, ie, a bit in the result is 1 only if both the corresponding bits in the factors are 1, 0 otherwise.

If used to combine relational values AND's factors should be either TRUE (–1) or FALSE (0). It returns –1 if both factors are TRUE, 0 otherwise.

ASN

Returns the arc-sine of a numeric value.

Syntax: <result> = ASN <factor>

Example:

```
PRINT ASN(opp/hyp)
```

ASN must be followed by a factor yielding a value between –1 and +1 inclusive. It returns the arc-sine of this value in radians as a real, in the range –PI/2 to +PI/2.

ATN

Returns the arc-tangent of a numeric value.

Syntax: <result> = ATN <factor>

Example:

```
ang = ATN(sin/cos)
```

ATN must be follwed by a factor yielding any numeric value. It returns the arc-tangent of this value in radians as a real, in the range –PI/2 to +PI/2.

COS

Returns the cosine of a numeric value.

Syntax: <result> = COS <factor>

Examples:

```
PRINT COS(ang)
cos = COS PI
```

COS must be follwed by a factor yielding any numeric value which is taken to be an angle in radians. It returns the cosine of this value which will be a real between –1 and +1 inclusive.

DEG

Converts radians to degrees.

Syntax: <result> = DEG <factor>

Example:

```
ang=DEG(ATN(a))
```

DEG must be followed by a factor yielding any numeric value which is taken to be an angle in radians. It returns the corresponding angle in degrees, which will be a real equal to 180*n/PI where n is the argument's value.

DIV

Returns the quotient of two integers.

Syntax: <result> = <factor> DIV <factor>

Example:

```
tens% = X% DIV 10
```

DIV must be both preceded and followed by a factor yielding an integer value, the second of which must be non-zero. It returns the integer quotient of the operands, rounded towards zero.

EOR

Returns the logical or bitwise exclusive-OR of two integer values.

Syntax: <result> = <factor> EOR <factor>

Examples:

```
oneflag% = flag1% EOR flag2%
bits = mask EOR val
```

EOR must be both preceded and followed by a factor yielding an integer value. It returns the value resulting from EORing the corresponding bits in these two values, ie, a bit in the result is 1 only if the corresponding bits in the factors are different, 0 otherwise.

If used to combine relational values EOR's factors should be either TRUE (–1) or FALSE (0). It returns –1 if one factor is TRUE and the other is FALSE, or 0 if they are both TRUE or both FALSE.

EXP

Returns the exponential of a numeric value.

Syntax: <result> = EXP<factor>

Example:

```
cosh(x) = (EXP(x) + EXP(-x))/2
```

EXP must be followed by a factor yielding any numeric value between the largest negative real available and approximately +88. It returns e (e = 2.718281828) raised to the power of this number, which is a real between 0 and the largest positive real available.

INT

Returns the integer part of a numeric value.

Syntax: <result> = INT <factor>

Examples:

```
x% = INT x
nearx% = INT (x + 0.5)
```

INT must be followed by a factor yielding any numeric value. It returns the nearest integer *less* than (not closer to zero than) or equal to this value (INT(-5.5) returns 6).

LN

Returns the natural logarithm of a numeric value.

Syntax: <result> = LN <factor>

Example:

```
PRINT LN 100
```

LN must be followed by a factor yielding any real value greater than zero. It returns the log to the base e (e=2.718281828) of this value.

LOG

Returns the logarithm of a numeric value.

Syntax: <result> = LOG <factor>

Example:

```
logx = LOG x
```

LOG must be followed by a factor yielding any real value greater than zero. It returns the log to the base 10 of this value which will be a real in the range –38 to + 38.

MOD

Returns the integer remainder of integer division.

Syntax: <result> = <factor> MOD <factor>

Example:

```
digits% = 10 MOD X%
```

MOD must be both preceded and followed by a factor yielding an integer value, the second of which must be non-zero. It returns the integer remainder of the left-hand factor divided by the right-hand one.

NOT

Returns the logical or bitwise NOT of an integer value.

Syntax: <result> = NOT <factor>

Examples:

```
IF NOT flag% PROCerror
inv%=NOT mask%
```

NOT must be followed by a factor yielding an integer value. It returns the value resulting from inverting the bits in this this value, ie, a bit in the result is 1 if the corresponding bit in the argument is 0 and vice versa.

If the argument is a truth value, NOT can be used in a logical statement to invert the condition. In this case, the truth value should only be one of the values TRUE (–1) or FALSE (0).

OR

Returns the logical or bitwise OR of two integer values.

Syntax: <result> = <factor> OR <factor>

Example:

```
either% = flag1% OR flag2%
```

OR must be both preceded and followed by a factor yielding an integer value. It returns the value resulting from ORing the corresponding bits in these values, i.e. a bit in the result is 1 if either or both of the corresponding bits in the factors is one, 0 otherwise.

If used to combine relational values ORs factors should be either TRUE (–1) or FALSE (0). It returns –1 if either or both of the factors are TRUE, 0 otherwise.

PI

Returns the value of π.

Syntax: <result> = PI

Example:

```
area = PI * rad * rad
```

PI returns the real constant 3.14159

RAD

Converts degrees to radians.

Syntax: <result> = RAD <factor>

Example:

```
sinx = SIN(RAD(ang))
```

RAD must be followed by a factor yielding any numeric value which is taken to be an angle in degrees. It returns the corresponding angle in radians, which will be a real equal to PI*n/180 where n is the argument's value.

RND

Returns a random number. RND takes two forms:

Syntax 1: <result> = RND

RND returns a four-byte signed random integer between –2147483648 and +2147483647.

Syntax 2: <result> = RND(<expression>)

Examples:

```
samerand = RND (0)
CIRCLE RND(1000),RND(1000),RND(diam%)
```

RND is followed by an expression yielding any numeric value. The value of the expression determines its action as follows:

<expression> < 0	This 're-seeds' the random number generator and RND returns its argument as a result. Re-seeding the generator with a given seed value always produces the same sequence of random numbers.
<expression> = 0	This uses the same seed as the last RND call and returns the same random number.
<expression> = 1	This returns a number in the range 0 to 1.
<expression> > 1	RND returns a random integer value between one and the integer value of the expression inclusive.

SGN

Returns the sign of a numeric value.

Syntax: <result> = SGN <factor>

Example:

```
number = ABS(x) * SGN(y)
```

SGN must be followed by a factor yielding any numeric value. It returns –1 if the value is negative, 0 if it is zero and +1 if it is positive.

SIN

Returns the sine of a numeric value.

Syntax: <result> = SIN<factor>

Example:

```
PRINT SIN(ang)
```

SIN must be followed by a factor yielding any numeric value which is taken to be an angle in radians. It returns the sine of this value which will be a real between –1 and +1 inclusive.

SQR

Returns the square-root of a numeric value.

Syntax: <result> = SQR<factor>

Example:

```
hyp = SQR (side1*side1 + side2*side2)
```

SQR must be followed by a factor yielding any positive numeric value. It returns the square-root of this value which will be a positive real.

TAN

Returns the tangent of a numeric value.

Syntax: <result> = TAN<factor>

Example:

```
opp=adj*TAN(RAD(ang))
```

TAN must be followed by a factor yielding any numeric value which is taken to be an angle in radians. It returns the tangent of this value.

9 : Strings

BASIC has very sophisticated string handling capabilities. They may contain between up to 255 characters enclosed in double quotes, for example:

```
"Hello"
"The quick brown fox jumps over the lazy dog"
```

String variable names are denoted by a terminating dollar '$', for example name$ and char$.

Strings may be 'added' together (concatenated), for example:

```
"Hello"+" "+"there"
```

produces the string "Hello there". However, '*', '-' and '/' are not legal string operators, but '=', '>' etc. can be used to compare strings. These compare the ASCII values of characters in corresponding positions in two strings until they find a difference, or both strings are exhausted.

This chapter describes the routines available for dealing with strings. These perform a wide range of operations including finding the length of a string, creating a sub-string from within a source string and converting between strings and numbers.

ASC

Returns the ASCII code of the first character in a string.

Syntax: <result> = ASC <factor>

Example:

```
ch% = ASC"A$"
```

ASC must be followed by a factor yielding a string of between 0 and 255 characters. It returns the ASCII code of the first character of the string, or –1 if the argument is a null string.

CHR$

Returns the character corresponding to an ASCII code.

Syntax: <result> = CHR$ <factor>

Example:

```
PRINT CHR$(code%)
```

CHR$ must be followed by a factor yielding an integer value in the range 0 to 255. It returns a string containing one character whose ASCII code is the value given.

INSTR(

Returns the position of a substring in a string.

Syntax : <result> = INSTR(<exp>,<exp>[,<exp>])

Example:

```
REPEAT a$=GET$:UNTIL INSTR("YyNn",a$) <> 0
```

INSTR$(may be followed by two or three expressions. The first two expressions must yield strings of between 0 and 255 characters, the third optional expression should yield an integer value in the range 0 to 255. INSTR$(searches the first string to find the next occurence of the substring given by the second argument. If the third argument is not present it starts the search at the beginning of the first string, otherwise it starts at the character given by the third value. It returns an integer in the range 0 to 255, this value being the position that the start of the substring was found in the main string. 0 means that the substring could not be found in the main string. This is always the case if the substring is longer than the main string. If the substring is a null string then the result will always be equal to the position of the start of the search.

LEFT$(

Returns or alters the left-hand characters of a string. LEFT$(comes in two forms:

Syntax 1: <result> = LEFT$(<exp>,[<exp>])

Examples:

```
REPEAT A$=LEFT$(A$):PRINT A$:UNTIL LEN(A$) = 1
C$ = LEFT$(A$+B$),LEN C$
```

LEFT$ is followed by one or two expressions, the first yielding a string of between 0 and 255 characters, the second optional one yielding an integer value in the range 0 to 255. It returns a string containing this number of characters from the left-hand side of the string, the default value being the length of the string –1. If the value is greater than the length of the string then all of the string is returned.

Syntax 2: LEFT$(<variable>[,<exp>]) = <exp>

Examples:

```
LEFT$(A$) = "XX"
LEFT$(A$,4) = B$
```

LEFT$ is followed by a string variable and an optional expression yielding an integer value. It replaces the characters in this string, starting from the left-hand character, by the string expression given on the right of the assignment. The maximum number of characters replaced is given by the second integer value if present, otherwise the replacement stops when either the end of the string variable or the end of the replacement string is reached, whichever is reached first.

LEN

Returns the length of a string.

Syntax: <result> = LEN <factor>

Example:

```
IF LEN A$ > 1 THEN A$ = RIGHT$(A$)
```

LEN must be followed by a factor yielding a string of between 0 and 255 characters. It returns the number of characters in this string.

MID$(

Returns or alters the middle characters of a string. MID$(comes in two forms which are detailed below:

Syntax 1: <result> = MID$(<exp>,<exp> [,<exp>])

Examples:

```
A$ = MID$(B$,5)
PRINT MID$("ABCDE",3,2);
```

MID$(is followed by two or three expressions, the first yielding a string of between 0 and 255 characters, the second, and optional third one yielding integer values. It returns a substring of the string given, starting from the position specified by the first integer value. If the second integer value is given, this specifies the maximum length of the substring, otherwise the substring contains the remainder of the source string.

Syntax 2: MID$(<variable>,<exp>[,<exp>]) = <exp>

Examples:

```
MID$(A$,n%) = "XXX"
MID$(A$,2,5) = MID$(B$,3,6)
```

MID$(is followed by a variable giving the name of a string and one or two expressions yielding integer values. It replaces the characters in this string, starting from the position given by the first integer value, by the contents of the string expression given on the right of the assignment. The maximum number of characters replaced is given by the second integer value if present, otherwise the replacement stops when either the end of the string variable or the end of the replacement string is reached, whichever is first.

RIGHT$(

Returns or alters the right-hand characters of a string. RIGHT$(comes in two forms:

Syntax 1: <result> = RIGHT$(<exp>[,<exp>])

Example:

```
PRINT RIGHT$("FRED",2)
```

RIGHT$(is followed by one or two expressions, the first yielding a string of between 0 and 255 characters, the second optional one yielding an integer value in the range 0 to 255. It returns a string containing this number of characters from the right-hand side of the string, the default value being one. If the value is greater than the length of the string, then all of the string is returned.

Syntax 1: RIGHT$(<variable>[,<exp>]) = <exp>

Example:

```
RIGHT$(A$,4) = "ABCD"
```

RIGHT$(is followed by a string variable and an optional expression yielding an integer value. It replaces the right-hand characters in this string, by the string expression given on the right of the assignment. The maximum number of characters replaced is given by the second integer value if present, otherwise the number changed is given by the lesser of the lengths of the string variable, or the length of the replacement string.

STR$

Returns the string representation of a number.

Syntax: <result> = STR$[~]<factor>

Example:

```
PRINT STR$(PI)
PRINT name$ + STR$(age%)
```

STR$ must be followed by a a factor yielding a numeric value. This factor may be preceded by a tilde '~' in which case the factor must give an integer rather than a floating point value. It returns a decimal (if the tilde is not present) or hex (if the tilde is present) string representation of the argument. @% may be used to determine the format of the string obtained.

STRING$(

Returns multiple copies of a string.

Syntax: <result> = STRING$(<expression1>,<expression2>)

Example:

```
line$=STRING$(30,"+-")+"+"
```

STRING$(must be followed by two expressions, the first yielding an integer value in the range 0 to 255, the second yielding a string of between 0 and 255 characters. It treats the integer value as the number of copies of the string required and returns a string of this number of concatenated copies of the source string. The integer copy value and length of the source string must be such that the resulting string contains between 0 and 255 characters.

VAL

Returns the numeric value of a decimal string.

Syntax: <result> = VAL <factor>

Example:

```
date=VAL(date$)
```

VAL must be followed by a factor yielding a string of between 0 and 255 characters. It returns the number that would have been read if the string had been typed in response to a numeric INPUT statement. The string is interpreted up to the first character which is not a legal numeric character.

10 : Screen Control

This chapter contains details on how to change screen mode in order to change graphics resolution, the number of text characters, or the number of colours available. It also details the keywords which set the text and graphics foreground and background colours, investigate the colour of particular pixels, clear the text or graphics windows and turn the cursor on or off.

In addition it documents the VDU command which is a general purpose command for controlling the VDU drivers.

CLG

Clears the graphics window.

Syntax: CLG

CLG clears the graphics window to the graphics background colour and moves the graphics cursor back to the graphics origin.

CLS

Clears the text window.

Syntax: CLS

CLS clears the text window to the text background colour and moves the text cursor to its 'home' position, normally the top left of the screen.

COLOUR (COLOR)

Sets the text colours or alters the palette settings. COLOUR comes in three different forms:

Syntax 1: COLOUR <expression> [TINT <expression>]

Example:

```
COLOUR 128+4
```

COLOUR is followed by an expression yielding an integer value in the range 0 to 255. If the value is in the range 0 to 127, COLOUR sets the text foreground colour. If the value is in the range 128 to 255 COLOUR sets the text background colour. The value MOD the number of colours in the current mode is used to give the 'logical' colour whose default settings are as given in the following table:

	2-colour	4-colour	16-colour
0	black	black	black
1	white	red	red
2	black	yellow	green
3	white	white	yellow
4	black	black	blue
5	white	red	magenta
6	black	yellow	cyan
7	white	white	white
8	black	black	flashing black-white
9	white	red	flashing red-cyan
10	black	yellow	flashing green-magneta
11	white	white	flashing yellow-blue
12	black	black	flashing blue-yellow
13	white	red	flashing magenta-green
14	black	yellow	flashing cyan-red
15	white	white	flashing white-black

In all modes colour 0 is black and colour 63 is white.

In the 256 colour modes 64 colours are available directly using COLOUR. The bit pattern of the value given determines the shade as follows:

bit 7	Controls foreground/background
bit 6	unused
bit 5	high blue
bit 4	low blue
bit 3	high green
bit 2	low green
bit 1	high red
bit 0	low red

COLOUR may also be followed by the TINT keyword. This is followed by an expression yielding an integer value in the range 0 to 255. It is only effective in 256 colour modes in which it selects the amount of white to be added to the colour. There are 4 distinct values 0 (darkest shade) – 3 (lightest shade).

Syntax 2: `COLOUR <exp>,<exp>`

Example:

```
COLOUR 2,5
```

COLOUR is followed by two expressions yielding integer values in the range 0 to 15. It assigns to the logical colour given by the first value the actual colour given by the second value. Then when the logical colour number is used the actual colour will be displayed. The actual colour numbers correspond to the default colours available in 16-colour modes, ie, eight steady colours and eight flashing colours.

Syntax 3: `COLOUR <exp>,<exp>,<exp>,<exp>`

Example:

```
COLOUR 3,255,255,255
```

COLOUR is followed by four expressions. The first should yield an integer value in the range 0 to 15. The next three should yield integer values in the range 0 to 255. The first value is the logical colour number. The next three specify the amounts of red, green and blue which are to be assigned to that logical colour. Only the top two bits of each are relevant.

GCOL

Sets the graphics colours and actions. GCOL comes in two forms:

Syntax 1: GCOL <exp1> [TINT <exp2>]

Example:

```
GCOL 2
```

GCOL is followed by an expression yielding an integer value in the range 0 to 255. If the value is in the range 0 to 127 GCOL sets the graphics foreground colour. If the value is in the range 128 to 255 GCOL sets the graphics background colour. The value MOD the number of colours in the current mode is used to give the 'logical' colour.

Syntax 2: GCOL <exp1>,<exp2> [TINT <exp3>]

Example:

```
GCOL ac%, col%
```

GCOL is followed by two expressions yielding integer values in the range 0 to 255. The first value determines the effect of future PLOT commands on the screen as follows:

0	Store the colour <expression2> on the screen
1	OR the colour <expression2> with the screen
2	AND the colour <expression2> with the screen
3	EOR the colour <expression2> with the screen
4	Invert the current colour, disregarding <expression2>
5	Don't affect the screen at all
6	AND NOT the colour <expression2> with the screen
7	OR NOT the colour <expression2> with the screen

The second value determines the colour that will combine with the screen for PLOT actions 0 to 5. If the value is in the range 0 to 127 GCOL affects the graphics foreground colour, if it is in the range 128 to 255 GCOL affects the graphics background colour. The value MOD the number of colours in the current mode is used to give the colour.

If the first value is in the range 16 to 23 the second value is ignored and the first extended colour fill (ECF) pattern is used instead. Similarly 32 to 29 uses the second ECF pattern, 48 to 55 uses the third ECF pattern, and 64 to 71 uses the fourth ECF pattern. 80 to 87 uses all four ECF patterns placed side by side. VDU23,2 to VDU23,5 are used to set the colour fill patterns. If the currently selected pattern is re-defined, it becomes active immediately.

MODE

Changes the display mode.

Syntax: MODE <expression>

Examples:

```
MODE 15
MODE m%+128
```

MODE must be followed by an expression yielding an integer in the range 0 to 255. It changes the screen mode to that value. The modes currently defined are in the range 0–20 and have the following characteristics:

Mode	Text	Graphics	Colours	Memory k
0	80x32	640x256	2	20
1	40x32	320x256	4	20
2	20x32	160x256	16	40
3	80x25	Text only	2	40
4	40x32	320x256	2	20
5	20x32	160x256	4	20
6	40x25	Text only	2	20
7	40x25	TELETEXT	16	80
8	80x32	640x256	4	40
9	40x32	320x256	16	40
10	20x32	160x256	256	80
11	80x25	Text only	4	40
12	80x32	640x256	16	80
13	40x32	320x256	256	80
14	80x25	Text only	16	80
15	80x32	640x256	256	160
16	132x32	Text only	16	132
17	132x25	Text only	16	132
18	80x64	640x512	2	40
19	80x64	640x512	4	80
20	80x64	640x512	16	160

Modes 18, 19 and 20 only function when a multi-sync monitor is used.

When a mode change is made the operating system:

1. Clears the screen to the current text background colour.
2. Sets the text and graphics windows to their defaults of the whole screen.
3. Homes the text cursor.
4. Moves the graphics cursor to (0,0).
5. Resets the logical-physical colour map to the default for the new mode.
6. Resets the colour fill patterns to their defaults for the new mode.
7. Sets the dot pattern for dotted lines to &AA.

OFF

Turns the cursor off.

Syntax: OFF

OFF turns the cursor off so that it is no longer visible.

ON

Turns the cursor on.

Syntax: ON

ON turns the cursor on so that it is visible. This is the default state.

POINT(

Returns the logical colour of a graphics pixel.

Syntax: <result> = POINT(<expression1>,<expression2>)

Example:

```
PRINT POINT (650,512)
```

POINT(must be followed by a pair of expressions yielding integer values in the range –32768 to +32767 and a closing bracket. It uses these as the x and y coordinates of a position and returns an integer in the range –1 to n, where n is one less than the number of logical colours in the current mode.

If the point specified lies outside the current graphics window, –1 is returned, otherwise, it is the logical colour of the point.

TINT

Returns the tint of a graphics pixel or sets the tint value. TINT comes in two forms as detailed below:

Syntax 1: <result> = TINT(<expression>,<expression>)

Example:

```
tint%=TINT(x%,y%)
```

TINT(is followed by a pair of expressions yielding integer values in the range –32768 to +32767 and a closing bracket. It uses these as the x and y coordinates of a position and returns the value –1 if the point is off the screen and &00, &40, &80 or &C0 otherwise. These give the amount of white tint at the point. In modes other than the 256 colour modes, the tint is &00.

Syntax 2: TINT <expression>,<expression>

Example:

```
TINT 1,&40
```

TINT is followed by a pair of expressions. The first should yield an integer values in the range 0 to 3, and the second an integer value in the range 0 to 255. It uses the first value to determine which colour is set as follows:

0	Sets the tint for the text foreground colour
1	Sets the tint for the text background colour
2	Sets the tint for the graphics foreground colour
3	Sets the tint for the graphics background colour

The top two bits of the second value determine the tint applied:

&00	Darkest (least white tint)
&40	:
&80	:
&C0	Lightest (most white tint)

VDU

Sends bytes to the VDU drivers.

Syntax: VDU [<exp>]{[, or ; or | [<exp>]} [; or |]

Examples:

```
VDU 7
VDU 23,10,2|
VDU 24,x1;y1;x2;y2;
```

VDU may be followed by a number of expressions yielding integer values. Each of these values is sent to the operating system VDU drivers. They should be followed by a comma (or nothing for the last expression) in which case the least significant byte (LSB) of the value is sent, or a semi-colon in which case the bottom two bytes are sent (low byte first) or a vertical bar in which case the LSB of the value is sent followed by nine zero bytes. As a maximum of 9 parameters are required by any VDU statement, the vertical bar ensures that sufficient parameters have been sent for any particular call. Any surplus are irrelevant, since VDU 0 does nothing.

The VDU codes are as follows:

VDU Code	Extra bytes	Meaning
0	0	Does nothing
1	1	Sends next character to printer only
2	0	Enables printer
3	0	Disables printer
4	0	Writes text at text cursor
5	0	Writes text at graphics cursor
6	0	Enables VDU driver
7	0	Generates BELL sound
8	0	Moves cursor back one character
9	0	Moves cursor on one space
10	0	Moves cursor down one line
11	0	Moves cursor up one line
12	0	Clears text area
13	0	Moves cursor to start of current line
14	0	Turns on page mode
15	0	Turns off page mode
16	0	Clears graphics area

VDU Code	Extra bytes	Meaning
17	1	Defines text colour
18	2	Defines graphics colour
19	5	Defines logical colour
20	0	Restores default logical colours
21	0	Disables VDU drivers or delete current line
22	1	Selects screen mode
23	9	Multi-purpose command
24	8	Defines graphics window
25	5	PLOT
26	0	Restores default windows
27	0	Does nothing
28	4	Defines text window
29	4	Defines graphics origin
30	0	Homes text cursor
31	2	Moves text cursor

11 : Graphics Output

Keywords are available which let you draw points, lines, rectangles, circles, and ellipses, flood fill enclosed areas or move/copy areas of the screen. Any co-ordinates needed by these have to be supplied relative to the graphics origin. By default this is it at the bottom left of the screen but it can be reset. The screen size is 1280 x 1024.

In addition there is a general PLOT command which gives a wider range of results but is harder to use. This keyword takes just a single pair of co-ordinates, if it requires more to define the shape it is trying to draw it uses the current and previous graphics cursor positions. For example it draws a triangle by joining the position given to both the current position that the graphics cursor is at and the position it was at previously. This means that you must pay careful attention to the position of the graphics cursor after each graphics operation otherwise future plots could produce unexpected results.

This section describes the keywords which draw graphics or affect the graphics origin or cursor position etc. For details of how to set the colours to be used and method of plotting etc. see the previous section 'Screen control'.

CIRCLE

Draws a circle. CIRCLE comes in two forms:

Syntax 1: CIRCLE <expression>,<expression>,<expression>

Example:

```
CIRCLE x%,y%,rad%
```

CIRCLE must be followed by three expressions which yield integer values in the range –32768 to +32768. It draws the outline of a circle using the first two values as the x and y co-ordinates of the centre and the third as the radius. The position of the graphics cursor is then updated to lie at the radial point, ie, the point on the circumference which is furthest to the right.

Syntax 2: CIRCLE FILL <expression>,<expression>,<expression>

Example:

```
CIRCLE FILL 640,512,100
```

CIRCLE must be followed by three expressions which yield integer values in the range –32768 to +32768. It draws a solid circle using the first two values as the x and y co-ordinates of the centre and the third as the radius. The position of the graphics cursor is then updated to lie at the radial point, ie, the point on the circumference which is furthest to the right.

DRAW

Draws a line to specified co-ordinates. DRAW comes in two forms:

Syntax 1: DRAW <expression>,<expression>

Example:

```
DRAW 1280,1024
```

DRAW must be followed by two expressions yielding integer values in the range –32768 to +32767. It uses these as the x and y co-ordinates of a new position and draws a line in the current graphics foreground colour between the current cursor position and this new position. It then updates the graphics cursor position to these co-ordinates.

Syntax 2: DRAW BY <expression>,<expression>

Example:

```
DRAW BY 32,64
```

DRAW BY must be followed by two expressions yielding integer values in the range –32768 to +32767. It uses these as the x and y offsets from the current graphics cursor and draws a line in the current graphics foreground colour between the current cursor position and the position offset by these values. It then updates the graphics cursor position to these co-ordinates.

ELLIPSE

Draws an ellipse. ELLIPSE comes in two forms:

Syntax 1: ELLIPSE <exp>,<exp>,<exp>,<exp>[,<exp>]

Example:

```
ELLIPSE x%,y%,smaj%,smin%
```

ELLIPSE is followed by four expressions which yield integer values in the range –32768 to +32768 and an optional fifth expression yielding a numeric value. It draws the outline of an ellipse using the first two values as the x and y co-ordinates of the centre, the third as the semi-major axis, the fourth as the semi-minor axis and the optional fifth value as the rotation in radians between the x-axis and the semi-major axis. If this value is absent then the angle is zero and the ellipse is axes-aligned.

Syntax 2: ELLIPSE FILL <exp>,<exp>,<exp>,<exp>[,<exp>]

Example:

```
ELLIPSE FILL 640,512,200,100,PI/4
```

ELLIPSE FILL is followed by four expressions which yield integer values in the range –32768 to +32768 and an optional fifth expression yielding a numeric value. It draws a solid ellipse using the first two values as the x and y co-ordinates of the centre, the third as the semi-major axis, the fourth as the semi-minor axis and the optional fifth value as the rotation in radians between the x-axis and the semi-major axis. If this value is absent then the angle is zero and the ellipse is axes-aligned.

FILL

Flood-fills an area.

Syntax: FILL <expression>,<expression>

Example:

```
FILL cenx%, ceny%
```

FILL must be followed by two expressions yielding integer values in the range –32728 to +32767. It uses these as the x and y co-ordinates of the start position. If this point is in a non-background colour then nothing happens. Otherwise it fills in all directions in the current foreground colour until it reaches either a non-background colour, the edge of the screen, or the edge of the graphics window.

LINE

Draws a line between two points

Syntax: LINE <exp>,<exp>,<exp>,<exp>

Example:

```
LINE 540,412,740,612
```

LINE must be followed by four expressions yielding integer values in the range –32768 to +32767. It uses the first two as the x and y co-ordinates of one end of the line, the second two as the x and y co-ordinates of the other end of the line and draws a line between them in the current graphics foreground colour. It then updates the graphics cursor position to the second pair of co-ordinates.

MOVE

Sets the position of the graphic cursor. MOVE comes in two forms:

Syntax 1: MOVE <expression>,<expression>

Example:

```
MOVE 0,0
```

MOVE must be followed by two expressions yielding integer values in the range –32768 to 32767. It uses these as the x and y co-ordinates of a position and moves the current graphics cursor to these co-ordinates.

Syntax 2: MOVE BY <expression>,<expression>

Example:

```
MOVE BY x%,y%
```

MOVE BY must be followed by two expressions yielding integer values in the range –32768 to 32767. It uses these as the x and y offsets and moves the graphics cursor position by these amounts.

ORIGIN

Moves the graphics origin.

Syntax: ORIGIN <expression>,<expression>

Example:

```
ORIGIN 640,512
```

ORIGIN must be followed by two expressions yielding integer values in the range –32768 to +32767. It uses these as x and y absolute co-ordinates and resets the graphics origin to be at this position. This position is then treated as (0,0) by subsequent graphics commands such as MOVE, LINE etc.

PLOT

Calls the operating system PLOT function.

Syntax: PLOT <expression>,<expression>,<expression>

Example:

```
PLOT 85,100,100 : REM Draw a triangle
```

PLOT must be followed by three expressions. The first should yield a integer value in the range 0 to 255, the second and third should yield integer values in the range –32768 to +32767. PLOT uses the first as the code to determine what to plot, and the second and third as the x and y co-ordinates of the position to use. The plot codes are as follows:

Values		Function
0 - 7	(&00 - &07)	Solid line including both end points
8 - 15	(&08 - &0F)	Solid line excluding final points
16 - 23	(&10 - &17)	Dotted line including both endpoints
24 - 31	(&18 - &1F)	Dotted line excluding final point
32 - 39	(&20 - &27)	Solid line excluding initial point
40 - 47	(&28 - &2F)	Solid line excluding both end points
48 - 55	(&30 - &37)	Dotted line excluding initial point
56 - 63	(&38 - &3F)	Dotted line excluding both end points
64 - 71	(&40 - &47)	Point Plot
72 - 79	(&48 - &4F)	Horizontal line fill (left & right) to non-background
80 - 87	(&50 - &57)	Triangle fill

Values		Function
88 - 95	(&58 - &5F)	Horizontal line fill (right only) to background
96 - 103	(&60 - &67)	Rectangle fill
104 - 111	(&68 - &6F)	Horizontal line fill (left & right) to foreground
112 - 119	(&70 - &77)	Parallelogram fill
120 - 127	(&78 - &7F)	Horizontal line fill (right only) to non-foreground
128 - 135	(&80 - &87)	Flood to background
136 - 143	(&88 - &8F)	Flood to foreground
144 - 151	(&90 - &97)	Circle outline
152 - 159	(&98 - &9F)	Circle fill
160 - 167	(&A0 - &A7)	Circular arc
168 - 175	(&A8 - &AF)	Segment
176 - 183	(&B0 - &B7)	Sector
184 - 191	(&B8 - &BF)	Block copy/move
192 - 199	(&C0 - &C7)	Ellipse outline
200 - 207	(&C8 - &CF)	Ellipse fill
208 - 215	(&D0 - &D7)	Graphics Characters
216 - 223	(&D8 - &DF)	Reserved for Acorn Expansion
224 - 231	(&E0 - &E7)	Reserved for Acorn Expansion
232 - 239	(&E8 - &EF)	Sprite Plot
240 - 247	(&F0 - &F7)	Reserved for USER programs
248 - 255	(&F8 - &FF)	Reserved for USER programs

In each block the offset from the base number has the following meaning:

0	Move cursor relative (to last graphics point visited)
1	Draw relative using current foreground colour
2	Draw relative using logical inverse colour
3	Draw relative using current background colour
4	Move cursor absolute
5	Draw absolute using current foreground colour
6	Draw absolute using logical inverse colour
7	Draw absolute using current background colour

An exception to this is for COPY/MOVE where the codes are as follows:

184	(&B8)	Move only, relative
185	(&B9)	Move rectangle relative
186	(&BA)	Copy rectangle relative
187	(&BB)	Copy rectangle relative
188	(&BC)	Move only, absolute
189	(&BD)	Move rectangle absolute
190	(&BE)	Copy rectangle absolute
191	(&BF)	Copy rectangle absolute

POINT

Plots a single point or moves the on-screen pointer. POINT can be used in three forms:

Syntax 1: POINT <expression>,<expression>

Example:

```
POINT FNx, FNy
```

POINT must be followed by two expressions yielding integer values in the range –32768 to +32767. It uses these as the x and y co-ordinates of a position and plots a point at this position in the current graphics foreground colour. It then updates the graphics cursor to these co-ordinates.

Syntax 2: POINT BY <expression>,<expression>

Example:

```
POINT BY x%,y%
```

POINT BY must be followed by two expressions yielding integer values in the range -32768 to +32767. It uses these as x and y offsets from the graphics cursor and plots a point at this position in the current graphics foreground colour. It then updates the graphics cursor to these co-ordinates.

Syntax 3: POINT TO <expression>,<expression>

Example:

```
POINT TO 640,512
```

POINT TO must be followed by two expressions yielding integer values in the range –32768 to +32767. It uses these as the x and y co-ordinates of a position and moves the on-screen pointer to this position provided that the pointer is not linked to the mouse position. If the pointer is linked to the mouse this command is ignored.

RECTANGLE

This command is used to draw a rectangle or copy/move a rectangular area of the screen RECTANGLE comes in four forms:

Syntax 1. RECTANGLE <exp1>,<exp2>,<exp3>,<exp4>

Example:

```
RECTANGLE 640,512,-100,-100
```

RECTANGLE is followed by four expressions yielding integer values in the range –32768 to +32767. It uses the first two as the x and y co-ordinates of one corner and the second two as the x and y offsets of the opposite corner and draws an axes-aligned rectangle in the current graphics foreground colour between the two co-ordinates. It then updates the graphics cursor to first set of co-ordinates.

Syntax 2: RECTANGLE FILL <exp1>,<exp2>,<exp3>,<exp4>

Example:

```
RECTANGLE FILL x%,y%,width%,height%
```

RECTANGLE FILL is followed by four expressions yielding integer values in the range –32768 to +32767. It uses the first two as the x and y co-ordinates of one corner and the second two as the x and y offsets of the opposite corner. It plots a solid axes-aligned rectangle in the current graphics foreground colour between the two sets of co-ordinates. It then updates the graphics cursor to the second set of co-ordinates.

Syntax 3:

RECTANGLE <exp1>,<exp2>,<exp3>,<exp4>TO<exp5>,<exp6>

Example:

```
RECTANGLE 640,512,100,100 TO 840,712
```

RECTANGLE is followed by four expressions yielding integer values in the range –32768 to +32767 followed by the keyword TO and two further similar expressions. It uses the first two values as the x and y co-ordinates of one corner of the source rectangle, the second two as the x and y offsets of the opposite corner and makes a copy of the rectangular screen area between the two position, using the third two values as the x and y co-ordinates of the position at which the bottom left corner of the copy is to be placed.

Syntax 4:

RECTANGLE FILL <exp1>,<exp2>,<exp3>,<exp4> TO <exp5>,<exp6>

Example:

```
RECTANGLE FILL x%,y%,w%,h% TO x%+w%,y%+h%
```

RECTANGLE FILL is followed by four expressions yielding integer values in the range –32768 to +32767 followed by the keyword TO and two further similar expressions. It uses the first two values as the x and y co-ordinates of one corner of the source rectangle, the second two as the x and y offsets of the opposite corner and moves the rectangular screen area between the two position, using the third two values as the x and y co-ordinates of the position at which the bottom left corner is to be placed. It replaces the old area with the current graphics background colour.

12 : Text Output

The position at which text appears on the screen when it is printed is determined by the current position of the 'text cursor'. This can be read and altered. In addition text can be restricted to appearing within only certain areas of the screen, these regions are known as 'text windows'.

The routine which actually performs the output is PRINT. This deals with both strings and numbers, and can be given a number of print formatters which control the way in which the text appears, for example numbers can be printed in right-justified columns.

This section lists the keywords directly involved with the output text. For details on how to affect the colour of the text and so on, see Chapter 10, 'Screen Control', for further details.

POS

Returns the x-co-ordinate of the text cursor.

Syntax: <result> = POS

Example:

```
curpos% = POS
```

POS returns a non-negative integer giving the x co-ordinate of the text cursor. The x co-ordinate is normally given relative to the left-hand edge of the text window. If the cursor direction has been altered using VDU 23,16, then it is given relative to the negative x edge of the screen which may be top, bottom, left or right.

PRINT

Prints information to the output stream(s).

Syntax: PRINT(<expression>)

Example:

```
PRINTTAB(3,3);"Hello"
PRINT NAME = ";name$'"AGE  = ";age%
```

PRINT may be followed by a number of string expressions, numeric expressions, and print formatters. By default, numerics are printed in decimal, right justified in the print field given by @% (see below). Strings are printed left justified in the print field. The print formatters have the following effects when printing numbers:

;	Prevents right justification of numbers in the print field. Sets numeric printing to decimal. Semi-colon stays in effect until a comma is encountered. If it is the last character of the PRINT statement, it prevents a new-line being printed
,	Right justifies numbers in the print field. This is the default print mode. Comma stays in effect until a semi-colon is encountered. If the cursor isn't at the start of a print field, it prints spaces to reach the next one.
~	Prints numbers as hexadecimal integers, using the current left/right-justify mode. Tilde stays in effect until a comma or semi-colon is encountered.
'	Prints a new line. Retains current left/right justify and hexadecimal/decimal modes.
TAB	Positions the cursor. Retains current left/right justify and hexadecimal/decimal modes. See keyword TAB(for more details.
SPC	Prints the given number of spaces. Retains current left/right-justify and hexadecimal/decimal modes. See keyword SPC(for more details.
<space>	Prints the next item. Retains current left/right justify and hexadecimal/decimal modes.

When strings are printed, the descriptions above apply except that hexadecimal mode does not affect strings. No trailing spaces are printed after a string unless it is followed by a comma. This prints enough spaces to move to the start of the next print field.

The format in which numbers are printed, and the width of print fields are determined by the value of the special system integer variable, @%. Each byte in the variable has the following meaning:

Byte four	If non-zero the STR$ function uses the print format determined by @% when converting its argument to a string. If zero it uses a default format. See also page 20.
Byte three	Ignored
Byte two	If this is in the range 0 – 9 floating point numbers will be printed in fixed format with this number of decimal places. This is provided that the total number of digits is not greater than 10 in which case the the number of decimal places is given by the lesser of byte two and 10 – the number of digits before the decimal point.
	If it is 10 or more, then floating point numbers are printed in general exponential format. Numbers less than 1E–1 or greater than or equal to 1E10 are given in the form n.nnnEn and numbers between these values are given in the form nnn.nnn.
Byte one	This gives the print field width in the range 0 to 255, for tabulating using commas. This field width will always be used, provided that the number fits within it. If the number is too large then it will be printed in the minimum possible number of places

The default value of @% is &A0A.

SPC

Generates spaces in text output.

Syntax: SPC <factor>

Example:

```
INPUT a$;SPC(4);b$
```

SPC must be followed by a factor yielding an integer in the range 0 to 255. It treats this as the number of spaces to be printed.

TAB(

Positions text cursor. TAB(is avaiable for use in two ways:

Syntax 1: TAB(<expression>)

Example:

```
PRINT TAB(10);a$;TAB(20);b$
```

TAB(is followed by an expression yielding an integer value in the range 0 to 255. It uses this value as the desired x co-ordinate of the cursor and prints spaces until this position is reached, generating a newline if the current position is equal to or greater than the position specified.

Syntax 2: TAB(<expression>,<expression>)

Example:

```
INPUT TAB(0,10)"Input your age",eggs%
```

TAB(is followed by two expressions yielding integer values. It uses these as the desired x and y co-ordinates of the cursor and moves the cursor to this position. If either of the co-ordinates lies outside the current text window TAB(has no effect.

VPOS

Returns the y-co-ordinate of the text cursor.

Syntax: <result> = VPOS

Example:

```
PRINT VPOS
```

VPOS returns a non-negative integer giving the y co-ordinate of the text cursor. The y co-ordinate is normally given relative to the top edge of the text window. If the cursor direction has been altered using VDU 23,16, then it is given relative to the negative y edge of the screen which may be top, bottom, left or right.

13 : Information Input

When a program requires data it can obtain it in a number of ways; it can take it from constants within itself, it can ask the user to supply it or it can read it from a file. The last of these methods is described in the 'File handling' section, the others are examined here.

Data can be supplied in blocks within the program. Each item of data can be read individually when required and the data pointer will then be stepped on to points at the next item. The data pointer can be specifically set so that items can be read more than once, or can be skipped over.

Alternatively the program can ask the user for input. This can range from a single character to a whole line of text input at the keyboard. It can even be in the form of buttons pressed on the mouse when the mouse pointer is at a particular position on the screen.

The relevant keywords are given below.

ADVAL

Returns data from an analogue port or buffer.

Syntax: <result> = `ADVAL` <expression>

Example:

```
charwait%=ADVAL(-1)
```

`ADVAL` must be followed by an expression yielding an integer value in the range –1 to –8. Note that it can return either the number of characters free, or the number of characters used, depending which buffer is specified. The integer value in the expression corresponds to the buffers as follows:

Value	Buffer name	Result Type	Range
–1	Keyboard (input)	No. of characters used	(0-31)
–2	RS-423 (input)	No. of characters used	(0-255)
–3	RS-423 (output)	No. of characters free	(0-191)
–4	Printer (output)	No. of characters free	(0-63)
–5	Sound 0 (output)	No. of bytes free	(0-15) 1 entry = 3 bytes
–6	Sound 1 (output)	No. of bytes free	(0-15) 1 entry = 3 bytes
–7	Sound 2 (output)	No. of bytes free	(0-15) 1 entry = 3 bytes
–8	Sound 3 (output)	No. of bytes free	(0-15) 1 entry = 3 bytes

DATA

Marks the position of data in the program.

Syntax: DATA [<constant>]{[,<constant>]}

Example:

```
DATA Fred,JIM,3,&78
```

DATA can be followed by any number of constants which may be either strings or numeric values in decimal, binary or hex. The constants are only evaluated when a READ statement requires them. The way in which they are interpreted depends on the type of variable in the READ statement. A numeric READ will expect a number, whereas a string READ will treat the data as a literal string. Leading spaces in the data item are ignored but trailing spaces (except for the last data item on the line) are counted. If it is necessary to have leading spaces, or a comma or quote in the data item, it must be put within quotes, for example:

```
DATA Fred,JIM,"  SHEILA  ",3.26,4,&AA,"Name, age"
```

If an attempt is made to execute a DATA statement, BASIC treats it as a REM. The DATA statement must be the first statement on a line.

GET

Returns a character code from the input stream.

Syntax: <result> = GET

Example:

```
REPEAT UNTIL GET=32
```

GET returns an integer in the range 0 to 255 which is the ASCII code of the next character in the buffer of the currently selected input stream (keyboard or RS–423). If no character is available, it will wait until one is entered. The character input is not echoed to the screen.

GET$

Returns a character from the input stream.

Syntax: <result> = GET$

Example:

```
REPEAT UNTIL GET$ = " "
```

GET$ returns a single character string which contains the next character in the buffer of the currently selected input stream. If no character is available it will wait until one is entered. The character is not echoed to the screen.

INKEY

Returns a character code from the input stream or the operating system code.

Syntax: <result> = INKEY <factor>

Example:

```
code%=INKEY(sec%*100)
REPEAT UNTIL INKEY(-99)
```

INKEY must be followed by a factor yielding an integer value in the range –256 to 32767. The result it returns depends upon this value as follows:

If the value is –256, INKEY returns a number indicating which version of the operating system is in use. If the value is in the range 0 – 32767 it is treated as a time limit in centi-seconds to wait for a key to be pressed. INKEY returns the ASCII code of the next character in the current input buffer – if one appears within the time limit, or –1 when the 'timeout' occurs.

If the value is in the range –255 to –1 it is treated as the 'negative INKEY code' of the key being interrogated. INKEY returns TRUE if the key is being pressed at the time of the call and FALSE otherwise. The negative INKEY codes are as follows:

key	INKEY number	key	INKEY number
f0	-33		
f1	-114		
f2	-115		
f3	-116		
f4	-21		
f5	-117		
f6	-118		
f7	-23		
f8	-119		
f9	-120		
A	-66		
B	-101		
C	-83		
D	-51		
E	-35		
F	-68		
G	-84		
H	-85		
J	-70		
K	-71		
L	-87		
M	-102		
N	-86		
O	-55		
P	-56		
Q	-17		
R	-52		
S	-82		
T	-36		
U	-54		
V	-100		
W	-34		
X	-67		
Y	-69		
Z	-98		

key	INKEY number	key	INKEY number
0	-40	keypad 0	-107
1	-49	keypad 1	-108
2	-50	keypad 2	-125
3	-18	keypad 3	-109
4	-19	keypad 4	-123
5	-20	keypad 5	-124
6	-53	keypad 6	-27
7	-37	keypad 7	-28
8	-22	keypad 8	-43
9	-39	keypad 9	-44
,	-103	keypad ,	-93
-	-24	keypad -	-60
.	-104	keypad .	-77
\	-121		
[	-57		
/	-105	keypad /	-75
]	-89		
^	-25		
_	-41		
:	-73		
;	-88		
@	-72		
ESCAPE	-113		
TAB	-97		
CAPS LOCK	-65		
SHIFT LOCK	-81		
CTRL	-2		
SHIFT	-1		
SPACE	-99		
DELETE	-90	keypad DELETE	-76
RETURN	-74	keypad RETURN	-61
COPY	-106		
cursor up	-58	centre mouse button	-11
cursor left	-26	left mouse button	-10
cursor right	-122	right mouse button	-12
cursor down	-42		
		keypad +	-59
		keypad #	-91
		keypad *	-92

INKEY$

Returns a character from the input stream.

Syntax: <result> = INKEY$ <factor>

Example:

```
A$=INKEY$(500)
```

INKEY$ must be followed by a factor yielding an integer value. The value is treated as a time limit in centi-seconds to wait for a key to be pressed. INKEY$ returns the next character in the current input buffer, if one appears within the time limit or a null string when the 'timeout' occurs.

INPUT

Returns a value or values from the input stream.

Syntax: INPUT<text><variable>

Examples:

```
INPUT text$,num%
INPUT TAB(10)"Name ",name$,'TAB(10)"Address ",addr$
```

INPUT may be followed by an optional prompt, which, if present, may be followed by a semi-colon or comma, which causes a ? to be printed out after the prompt. These are followed by a list of variable names of any type, separated by commas. After the last variable, the whole sequence may be repeated, separated from the first by a comma. In addition the position of prompts may be controlled by the SPC, TAB(and ' print formatters (see PRINT).

INPUT reads values from the current input stream, assigning each in turn to any variable names it has been given, until all variables have been assigned values. It skips leading spaces and treats commas as marking the end of input for the current item.

INPUT LINE and LINE INPUT

Example:

```
INPUT LINE "Enter a sentence" sen$
```

INPUT LINE may be followed by an optional prompt, which, if present, may be followed by a semi-colon or comma, which causes a ? to be printed out after the prompt. These are followed by a list of variable names of any type, separated by commas. After the last variable, the whole sequence may be repeated, separated from the first by a comma. In addition the position of prompts may be controlled by the SPC, TAB(and ' print formatters (see PRINT).

INPUT LINE reads values from the current input stream assigning each in turn to any variable names it has been given, until all variables have been assigned values. If the input variable is a string, all the input is read into the variable, including leading and trailing spaces and commas. If the input variable is numeric, only a single value will be selected from the input line.

MOUSE

Controls and returns information from the mouse. MOUSE may be used in seven different forms:

Syntax 1:
MOUSE <integervar1>,<integervar2>,<integervar3>,[<integervar4>]

Example:

```
MOUSE xpos%,ypos%,button%
```

MOUSE is followed by three or four integer variables. It assigns the x and y positions of the mouse as signed 16-bit integer values to the first two variables and an integer in the range 0 to 7 to the third which gives the status of the mouse buttons as follows:

0	No buttons pressed
1	Right button pressed only
2	Middle button pressed only
3	Middle and right buttons pressed
4	Left button pressed only
5	Left and right buttons pressed
6	Left and middle buttons pressed
7	All three buttons pressed

The optional fourth parameter will be assigned the value of TIME.

Syntax 2: MOUSE ON [<expression>]

MOUSE ON may be followed by an expression yielding an integer value which must (currently) lie in the range of one to four. The command causes the mouse pointer to be displayed. If the optional integer expression is also supplied the pointer shape corresponding to this number will be used.

One default shape is provided by the operating system, the other three requiring user definition before use. There is no BASIC keyword for mouse pointer definition, so a call to the operating system is required.

Syntax 3: MOUSE OFF

MOUSE OFF turns off the mouse pointer.

Syntax 4: MOUSE COLOUR <exp>,<exp>,<exp>,<exp>

Example:

```
MOUSE COLOUR col%,red%,green%,blue%
```

MOUSE COLOUR is followed by four expressions. The first should yield an integer value in the range 0 to 15. The next three should yield integer values in the range 0 to 255. The first value is the logical mouse colour number. The next three specify the amounts of red, green and blue which are to be assigned to that logical colour. Only the top two bits of each are relevant.

Syntax 5: MOUSE TO <expression>,<expression>

Example:

```
MOUSE TO 640,512
```

MOUSE TO is followed by two expressions yielding integer values in the range –32768 to +32767. It treats these values as x and y co-ordinates and moves the mouse pointer to that position.

Syntax 6: MOUSE STEP <expression>[,<expression>]

Example:

```
MOUSE STEP 3,2
```

MOUSE STEP may be followed by either one or two expressions yielding integer values. It controls the speed of movement of the mouse. If one expression is given it is used as a multiplier for both the x and y movements. If two expressions are given, the first is used for x and the second for y.

Syntax 7: MOUSE RECTANGLE <exp>,<exp>,<exp>,<exp>

Example:

```
MOUSE RECTANGLE 640,512,1023,1279
```

MOUSE RECTANGLE is followed by four expressions yielding integer values in the range –32768 to +32767. It uses the first two as the x and y co-ordinates of the bottom left-hand corner of a rectangle, and the second two as the top right-hand corner of the rectangle. This rectangle sets the limits outside which the mouse cannot move. If the mouse pointer is outside the box when this command is given, it will be moved to the nearest point within it.

READ

Returns information from a DATA statement.

Syntax: READ [<variable>],[<variable>] etc

Examples:

```
READ a,b,c
READ string$,int%,float
```

READ may be followed by any number of variable names. The type of each variable should correspond to items in the DATA statement being read. For each variable, READ takes a value from the DATA statement and assigns it to the variable, and then moves the data pointer on to the start of the next item of data.

RESTORE

Sets the DATA pointer.

Syntax: RESTORE [<expression>]

Examples:

```
RESTORE
RESTORE 1000
```

DATA may be followed by an expression which yield a value in the range 0 to 65279, ie, a line number. If the expression is not given, RESTORE resets the DATA pointer to the first DATA statement in the program. If an expression is given, RESTORE sets the DATA pointer to the first item of data on or after the line number (as it was in the source program) specified.

14: File Handling

Data can be written to a file and then recalled again later when required. To do either of these operations you first need to open a file so that it can be written to or read from. When a file is opened, it has a specific number allocated to it this is called the 'channel number'. This number is used to communicate with the file. The advantage of this technique is that allows several files to be open at once and data to be written to or read from each file selectively.

Data can be written to or read from files a byte at a time, or as complete strings or numbers. The position at which the next object will be written or read is determined by the 'file pointer'. This pointer is incremented automatically as data is written or read but it can also be set explicitly. In addition the extent of a file (length) can be controlled and its current size determined.

The BASIC keywords used to perform these handling tasks are described in this chapter.

BGET#

Returns the next byte from a file.

Syntax: <result> = BGET# <factor>

Example:

```
char%=BGET#(channel%)
```

BGET# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. It returns the ASCII code of the character read (at position PTR#) from the file in the range 0–255. PTR# is then updated to 'point to' the next character in the file.

If the last character in the file has been read, EOF# for the channel will be TRUE. In this case the next BGET# will return an undefined value and the one after that will produce an 'EOF'error.

BPUT#

Writes a byte to a file.

Syntax: `BPUT# <factor>,<expression>`

Examples:

```
BPUT#channel,a$
BPUT#chan,char%
```

BPUT# must be followed by a factor yielding an integer, which is a channel number (as returned by one of the OPEN functions) and an expression.

If the expression yields a floating point value it is truncated to an integer in the range 0 to 255 and is treated as the ASCII code of a character. BPUT# then sends the corresponding character to the file.

If the expression yields a string, BPUT# sends the ASCII codes of all the characters in the string to the file. Then, unless the string is terminated by a semicolon ';' it adds a carriage return (ASCII value 13) to the end of the string.

In both cases PTR# is updated automatically to point to the next character to be written. If the end of the file is reached, the length (EXT#) will increase too. It is only possible to use BPUT# with files opened with OPENUP and OPENOUT files, not those opened with OPENIN.

CLOSE#

Closes an open file.

Syntax: `CLOSE# <factor>`

Example:

```
CLOSE# channel%
```

CLOSE# must be followed by a factor yielding either zero or an integer value which is a channel number as returned by one of the OPEN functions.

If the value is zero, CLOSE# closes all open files on the current filing system. If the factor is not zero only the file with the channel number specified will be closed.

Closing a file ensures that its contents are properly updated on whatever medium is being used. This is necessary as a certain amount of buffering is used to make the transfer of data between computer and mass-storage devices more efficient. Closing a file opened for update therefore performs two important tasks. It ensures that all data has been written and it updates the file length if this has changed. In all cases it releases the buffer for use by another file.

EOF#

Returns a value indicating if the end of a file has been reached.

Syntax: <result> = EOF# <factor>

Example:

```
IF EOF# chan1% PRINT "End of file"
```

EOF# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. It returns TRUE if the last character in the specified file has been read, and FALSE otherwise.

EXT#

Returns or controls the length – the extent – of an open file. EXT# may be used in two ways:

Syntax 1: <result> = EXT# <factor>

Example:

```
IF EXT#chan% = max% PRINT "File full":CLOSE#file
```

EXT# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. It returns an integer in the range 0 to 2147483648 which specifies the length of the file in bytes.

Syntax 2: EXT# <factor> = <expression>

Example:

```
EXT# chan%=EXT#chan% - 100
```

EXT# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions.

It is assigned the value of an expression which must yield a positive integer value. This is used as the desired extent of the file. EXT# sets the file length to this value. The main use of EXT# is to shorten a file. A file may be lengthened by using PTR#.

GET$#

Returns a string from a file.

Syntax: <result> = GET$# <factor>

Example:

```
string$ = GET$# chan%
```

GET$# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. It returns a string of characters until a linefeed (CHR$10), carriage return (CHR$13), a null character (CHR$0), or the end of the file is encountered or the maximum of 255 characters is reached. It then updates PTR# to 'point to' the next character in the file. If the last character in the file has been read, EOF# for the channel will be TRUE.

INPUT#

Obtains a value or values from a file.

Syntax: INPUT# <factor>{[, or ; <variable>]}

Example:

```
INPUT# chan1%,name$,age%
```

INPUT# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. This factor can be followed by any number of variables of any type. For each variable INPUT# reads data from the file and assigns it to the variable.

OPENIN

Opens a file for input only and returns its channel number

Syntax: <result> = OPENIN <factor>

Example:

```
chan% = OPENIN("DATA")
```

OPENIN must be followed by a factor which yields a string evaluating to a valid filename for the filing system in use. It returns an integer acting as a channel number for the file. The exact value depends upon the filing system and how many files are already open. It is zero if the file wasn't found. The file is opened for input only.

OPENOUT

Opens a new file for output and returns its channel number.

Syntax: <result> = OPENOUT <factor>

Example:

```
data% = OPENOUT("$.files.data")
```

OPENOUT must be followed by a factor which yields a string evaluating to a valid filename for the filing system in use. It returns an integer acting as a channel number for the file. If the file doesn't already exist, a new one is created. If a file of the same name already exists, that file is first deleted and a new one is created. The file is opened for output only.

OPENUP

Opens a file for input and output and returns its channel number.

Syntax: <result> = OPENUP <factor>

Example:

```
file% = OPENUP("records")
```

OPENOUT must be followed by a factor which yields a string evaluating to a valid filename for the filing system in use. It returns an integer acting as a channel number for the file. If the file doesn't already exist, then a new one is created. If a file of the same name does exist then that file is opened for input and output.

PRINT#

Prints information to an open file.

Syntax: PRINT# <factor>{[,<expression>]}

Example:

```
PRINT#file,name$,age%,height
```

PRINT# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. This factor can be followed by any number of expressions yielding any type. PRINT# evaluates each expression and sends it to the specified file with the corresponding type information:

- Integers are written as &40 followed by the two's complement representation of the integer in four bytes, least significant byte first.
- Reals are written by default as &F0 followed by four bytes as a single precision floating point number, the least significant byte being sent first. see page 20, 'Floating Point in Files' for the effect of @% on the format used.
- Strings are written as &00 followed by a one byte count for the length of the string, followed by the characters in the string in reverse order.

PTR#

Returns or controls the file pointer. PTR# may be used in one of two different ways:

Syntax 1: <result> = PTR#<factor>

Example:

```
PRINT PTR# chan%
```

PTR# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions. It returns an integer in the range 0 to 2147483648 which specifies the position relative to the start of the file of the next byte to be read or written.

Syntax 2: PTR# <factor> =<expression>

Example:

```
PTR# chan% = record_len%
```

PTR# must be followed by a factor yielding an integer value which is a channel number as returned by one of the OPEN functions.

It is assigned the value of an expression which must yield a positive integer value which is used as the desired position of the pointer in the file. PTR# then sets the pointer to this position.

15 : Sound

This chapter describes all the commands which enable the programmer to create sounds or alter the way in which they are produced. The Archimedes has eight different sound channels. The amplitude, pitch, duration and stereo position of the notes produced on each can be programmed and the output from the channels can be synchronised.

BEAT

Returns the current beat value.

Syntax: <result> = BEAT

Example:

```
PRINT BEAT
```

BEAT returns an integer giving the value of the beat counter. This counter counts from zero to the number given by BEATS at a rate determined by TEMPO and then resets to zero. All synchronisation is performed relative to the last beat counter reset.

BEATS

Returns or alters the beat counter. BEATS may be used in two ways:

Syntax 1: <result> = BEATS

Example:

```
PRINT BEATS
```

BEATS returns the current beat counter.

Syntax 2: BEATS <expression>

Example:

```
BEATS 200
```

BEATS is followed by an expression which yields an integer value. It uses this value as the limit up to which the beat counter is to count before it resets. The beat counter is used in conjunction with the SOUND and TEMPO statements to synchronise sound outputs on different sound channels.

ENVELOPE

Defines the volume and pitch of a sound .

Syntax: ENVELOPE <exp1>,<exp2>,...,<exp14>

ENVELOPE is followed by 14 expressions which yield integer values. The range of values and purpose of each of the 14 expressions depends on their position in the sequence as shown below.

The parameters fall into three categories, and the table is similarly divided to make reference easier.

Part one is parameters one and two, identifying the envelope and the rate of change of the sounds produced. Part two controls the pitch, and occupies parameters three to nine inclusive. Pitch is deemed to consist of three parts, so the parameters are in two groups of three. Part three controls the amplitude, and uses the last six parameters. Amplitude is defined in four sections, known as attack, decay, sustain and release phases. The first four parameters relate directly to these phases, while the last two define the sound level to be achieved at the end of each of the first two phases.

Parameter	Range	Meaning
1	1 – 4	The envelope number
2	0 – 255	See below
	bits 0 to 6	
	0 – 127	Length of time steps in centiseconds
	bit 7	Auto repeat the pitch envelope when clear
	bit 7	No auto repeat when set
Pitch		
3	–128 – 127	Change of pitch per time step, part
1 4	–128 – 127	Change of pitch per time step, part
2 5	–128 – 127	Change of pitch per time step, part
3 6	0 – 255	Number of steps in part 1
7	0 – 255	Number of steps in part 2
8	0 – 255	Number of steps in part 3

Amplitude

9	–127 – 127	Change of amplitude per time step during attack phase
10	–127 – 127	Change of amplitude per time step during decay phase
11	–127 - 127	Change of amplitude per time step during sustain phase
12	-127 – 127	Change of amplitude per time step during release phase
13	0 – 126	Target amplitude, end of attack phase
14	0 – 126	Target amplitude, end of decay phase

NB: This information has been included for completeness, because the compiler supports the ENVELOPE statement, as does BASIC V. However, at the time of writing, although ENVELOPE can be included in programs without error, it has absolutely no effect on sound production.

It must be assumed that ENVELOPE is included in BASIC V only for reasons of compatibility with earlier versions of BBC BASIC. Users should note that, although a later release of the operating system could support ENVELOPE, no known version, including the not yet released (at November 1988) version 2.0, does so.

SOUND

Generates a sound or suppresses/allows subsequent sound generation. SOUND may be used in three ways as detailed below:

Syntax 1: SOUND ON

SOUND ON allows sounds to be produced by subsequent use of the third form of the SOUND statement. This is the default state.

Syntax 2: SOUND OFF

SOUND OFF prevents subsequent SOUND statements from producing a noise.

Syntax 3: SOUND <exp>,<exp>,<exp>,<exp>

Examples:

```
SOUND 1,-15,255,10
SOUND &102,&140,&2400,200
```

SOUND is followed by four expressions yielding integer values. It uses the first as the channel number, the second as the amplitude, the third as the pitch and the fourth as the duration and makes the corresponding sound.

Channel Number

C is an integer in the range 0 to 32767. It should be regarded as a four-digit hex number, which can be written as: &HSFC. The meanings of these four parts are, in reverse order:

C The sound channel in the range 0 to 7.

F The 'flush' part inthe range 0 to 1.

A value of zero means that the note is added to the sound queue for the appropriate channel as normal.

A value of one means that the sound queue for channel C is 'flushed' before sounding the note, so that any note sounding on channel C, and any waiting in the queue, are removed and the sound generated with the 'flush' bit is sounded immediately.

S The synchronise part in the range 0 to 2.

A value of zero means that the note specified is to be played as soon as the previous note in the queue has been playing for the time specified in its duration part (see below). This is the normal state.

A value of one means that the note to be played until there is another note ready (on another channel's queue) with its synchronise part set to one. When such a note becomes available, both notes will be sounded together.

A value of two means that the note is not to be sounded until two other notes, also with synchronisation values of two, are present on other queues.

H In the range 0 to 1

A value of zero means that the note start playing as soon as the current note on the queue has been playing for the time specified in its duration part.

A value of 1 means that the new note to be queued in the normal way, but not sounded, ie, the pitch and amplitude parts are ignored. In addition, it won't truncate the release part of a note already playing.

Amplitude

This is an integer in one of two different ranges:

> In the range –15 to 0 it is a simple volume (amplitude), –15 being the loudest and zero being the quietest (ie no sound).
>
> In the range 256 (&100) to 511 (&1FF) it is a logarithmic volume, a difference of 16 providing a doubling or halving of the volume.

Pitch

> In the range 0 to 255 the note middle C has a pitch value of 53 and a difference of 48 corresponds to a difference in pitch of one octave, ie, there are four pitch values per semi-tone.
>
> In the range 256 (&100) to 32767 (&7FFF), the note middle C has a pitch value of &4000 and a difference of &1000 corresponds to a difference in pitch of one octave.

Duration

This is an integer in the range 0 to 255. It gives the duration of the note in twentieths of a second. A value of 255 produces a continuous sound.

STEREO

Sets the stereo position of a sound channel.

Syntax: STEREO <expression>,<expression>

Example:

```
STEREO chan%,64
```

STEREO must be followed by two expressions, the first yielding an integer in the range 1 to 8, the second an integer in the range –127 to 127. It uses the first value as the number of the channel whose stereo position is to be altered. The value must be less than or equal to the number of active channels. The channels affected are as follows

Active channels	Stereo Effect
1	all channels
2	the channel specified and every alterate one above it
4	the channel specified and the one 4 above it
8	the channel specified only

It assigns to these channel(s) the stereo position determined by the second value; –127 meaning that the sound is fully to the left, 0 meaning that it is in the middle and 127 meaning that it is fully to the right.

TEMPO

Returns or alters the beat counter rate. TEMPO comes in two forms:

Syntax 1: <result> = TEMPO

Example:

```
PRINT TEMPO
```

TEMPO returns the current tempo.

Syntax 2: TEMPO <expression>

Example:

```
TEMPO &4000
```

TEMPO is followed by an expression which yields an integer value which is treated as a hexadecimal fractional number, the three least significant digits giving the fractional part. It uses this value as the tempo to use to determine when to update the beat counter. A value of &1000 corresponds to a tempo of one beat per centi-second. Doubling this value causes the tempo to double (two tempo beats per centi-second) and halving the value halves the tempo (half a beat per centi-second).

VOICE

Binds a named voice to a given channel.

Syntax: VOICE <expression>,<expression>

Example:

```
VOICE 1,"Wave-Synth"
```

The first expression is the channel number, and the second expression is the named voice.

VOICES

Specifies the number of sound channels to be used.

Syntax: VOICES <expression>

Example:

```
VOICES 4
```

VOICES must be followed by an expression which yields an integer value in the range 1 to 8. The value of expression is rounded up (if necessary) to one, two, four or eight, and this number of sound channels is activated.

16 : The Operating System

Four keywords are provided to access the numerous Operating System routines and for executing separate, user defined, machine code routines.

CALL

Executes a machine code routine.

Syntax: CALL<exp>{[,<exp>]}[TO <var>{[,<var]}][;<var>]

CALL must be followed by an expression yielding an integer value. It uses this as the address of the machine code routine to call. It may also be followed by up to eight further expressions yielding either integer or string values. It passes these to the routine via the registers R0 – R7. If the value is an integer, it places it directly in a register. If the value is a string, it stores it in memory terminated by a null character and places the pointer to it in the register.

When supplying integer data to be placed in the registers, although it must be presented in sequence, only required values other than zero need be specified. Gaps in the expression sequence can be indicated by just a comma entry. these are evaluated as zero during the call. For example:

```
        CALL code% A%,B%,,,C%
```

would call a routine at address code%. The effect of the appended variable names and commas is to place the contents of A% in R0, the contents of B% in R1, zero in R2 and R3, and the contents of C% in R4. In this case R2 and R3 are set implicitly to zero, since the commas indicate two entries between B% and C%. Also, in this example expressions for R5, R6 and R7 have not been supplied. Note that any register contents not explicitly or implicitly set must be regarded as undefined.

The expressions can be followed by the keyword TO and a list of up to eight variables. CALL assigns the values in the registers R0 – R8 to these variables on return from the routine. If the variable to be assigned is numeric, the integer in the register is converted to an appropriate format and stored in it. If the variable to assign to is a string, the register is treated as a pointer to a string terminated by 0, 10 or 13 and this string is assigned to the variable. The strings given on input can be overwritten, but should not be extended.

CALL may be ended by a semicolon and a further numeric variable. On return from the routine it assigns the processor flag bits to this variable if it was supplied.

Note that if the address used is a constant value which corresponds to the address of one of the 6502 operating system routines, then a warning will be generated at compile time.

OSCLI

Passes a string to the operating system command line interpreter.

Syntax: OSCLI <expression>

Examples:

```
OSCLI "CAT"
OSCLI "SPOOL "+name$
```

OSCLI must be followed by an expression yielding a string of between 0 and 255 characters. It passes this string to the operating system command line interpreter for execution.

SYS

Calls operating system routines.

Syntax: SYS<exp>{[,<exp>]}[TO<var>[,<var>]}][;<var>]

Examples:

```
SYS 0,ASC"X"
REM Write the character X to the screen

SYS 4 TO char%
REM Read the ASCII value of a character input
```

SYS must be followed by an expression yielding either an integer or string value. It uses this as the identifier of the operating system routine to be called. It may also be followed by up to eight further expressions yielding either integer or string values. These are passed to the routine via the registers R0 – R7. If the value is an integer, it is placed directly in a register. If the value is a string, it is stored in memory, terminated by a null character, and the pointer to the memory location is placed in the register.

When supplying integer data to be placed in the registers, although it must be presented in sequence, only required values other than zero need be specified. Gaps in the expression sequence can be indicated by just a comma entry. these are evaluated as zero during the call. For example:

```
CALL code% A%,B%,,,C%
```

would call a routine at address code%. The effect of the appended variable names and commas is to place the contents of A% in R0, the contents of B% in R1, zero in R2 and R3, and the contents of C% in R4. In this case R2 and R3 are set implicitly to zero, since the commas indicate two entries between B% and C%. Also, in this example expressions for R5, R6 and R7 have not been supplied. Note that any register contents not explicitly or implicitly set must be regarded as undefined.

The expressions can be followed by the keyword TO and a list of up to eight variables. SYS assigns the values in the registers R0 – R8 to these variables on return from the routine. If the variable to be assigned is numeric, the integer in the register is converted to an appropriate format and stored in it. If the variable to be assigned is a string, the register is treated as a pointer to a string terminated by 0, 10 or 13 and this string is assigned to the variable. The strings given on input can be overwritten, but should not be extended.

SYS may be ended by a semicolon and a further numeric variable. On return from the routine it assigns the processor flag bits to this variable if it was supplied.

SYS provides access to all the routines supplied by the operating system and modules. These cover a wide range of functions including inputting and outputting characters, error handling, sprite manipulation, etc. Details of these operating system routines are given in the Acorn 'Programmer's Reference Manual'.

USR

Returns a value from a machine code routine.

Syntax: <result> = USR (<exp> {[,<exp>]})

USR must be followed by an expression yielding an integer value. It uses this as the address of the machine code routine to be called. It may also be followed by up to eight further expressions yielding either integer or string values. It passes these to the routine via the registers R0 – R7. If the value is an integer, it places it directly in a register. If the value is a string, it stores it in memory terminated by a null character and places the pointer to it in the register.

When supplying integer data to be placed in the registers, although it must be presented in sequence, only required values other than zero need be specified. Gaps in the expression sequence can be indicated by just a comma entry. these are evaluated as zero during the call. For example:

```
            Res%=USR(code%,A%,B%,,,C%)
```

would call a routine at address code%. The effect of the appended variable names and commas is to place the contents of A% in R0, the contents of B% in R1, zero in R2 and R3, and the contents of C% in R4. In this case R2 and R3 are set implicitly to zero, since the commas indicate two entries between B% and C%. Also, in this example expressions for R5, R6 and R7 have not been supplied. Note that any register contents not explicitly or implicitly set must be regarded as undefined.

USR returns the value in R0 on termination of the routine.

Note that if the address used is a constant value which corresponds with the address of one of the 6502 operating system routines, then a warning will be generated at compile time.

17 : Errors

Although syntax errors in a program will be found during compilation, errors can occur at run-time due to the program not being able to cope with the data it has been given. For example, by attempting to divide a numeric value by a variable or expression which evaluates to zero. This chapter describes the keywords available for trapping, investigating and handling any errors which occur.

ERL

Returns zero.

Syntax: `ERL`

ERL, under the interpreter, returns the line number on which the last error was found. However, the compiler's object code has no concept of line numbers. Therefore, ERL has been programmed to always return zero.

ERR

Returns the last error number.

Syntax: `ERR`

Example:

```
IF ERR=18 PRINT "Division by zero"
```

ERR returns a four-byte signed integer giving the number of the last error. The numbers of the errors are as follows:

Err	Message	Reason
0	Stopped	A STOP statement has been executed
0	Stack overflow	The stack has filled up and no more procedures can be called
0	Program terminated	The program has terminated
6	Type mismatch	The data type of an object was unacceptable

Err	Message	Reason
15	Array subscript out of range	The subscript used exceeds the maximum for the dimension of the array or was negative
41	Line number not found	The line number referes to by GOTO, GOSUB or RESTORE does not exits
40	On range	The argument of an ON..GOTO/GOSUB or PROC structure was higher than the number of actions given or was less than 1
42	Out of data	READ could find no more DATA
256	Version 0.2 08/07/88	The compiler version and date. This is not reported
19	String too long	The length of a string would have exceeded 255 characters
257	Bad alignment	An assembler instruction would have been placed at a non-aligned address
258	Bad immediate constant	The constant cannot be fitted within the bit field of the instruction
259	Branch target not aligned	The offset given to a branch or BL was not a whole number of 32-bits
260	Unknown token	The compiler has come across a token which it does not understand. This should only occur at compile time.
261	Bad number	A number was malformed
18	Division by zero	An attempt to do integer division by zero was made. (floating point division gives an exception from the Floating Point Emulator with its own error number and message)
17	Escape	An escape condition has been detected.

ERROR

Generates an error.

Syntax: ERROR <expression>,<expression>

Example:

```
ERROR ERR,REPORT$
```

ERROR must be followed by two expressions, the first yielding a four-byte signed integer value, the second yielding a string of between 0 and 255 characters. It treats the integer value as an error number and the string as the error message associated with this number and generates the error described.

ON ERROR

Error handler.

Syntax: ON ERROR <statements>

Examples

```
ON ERROR PROCerror(ERR)
ON ERROR GOTO 1000
```

ON ERROR can be followed by a number of BASIC statements. When an error occurs, these statements are executed.

REPORT

Prints the message of the last error encountered.

Syntax: REPORT

REPORT prints the message of the last error which occurred in the program.

REPORT$

Returns the last error message.

Syntax: <result> = REPORT$

Example:

```
PRINT REPORT$
```

REPORT$ returns a string containing the textual description of the last error which occurred in the program.

18 : Pseudo Variables

The ‘pseudo-variables’ are special variables used by BASIC to hold values describing the current state of certain aspects of the computer, for example, how memory is being used, what date the real-time clock is set to, and so forth. The values they return can be used in expressions like normal variables, for example:

```
PRINT (END + 25.4)/FNdiv
a% = PAGE ?100
```

However, most cannot be assigned to.

END

Returns the top of the heap.

Syntax: <result> = END

Examples:

```
freestack% = EXT - END
heapsize% = END - LOMEM
```

END returns the address of the top of the heap.

EXT

Returns the current stack pointer.

Syntax: <result> = EXT

Examples:

```
stacksize% = HIMEM - EXT
freestack% = EXT - END
```

EXT returns the address of the top of the stack.

FALSE

Returns the logical value 'FALSE'.

Syntax: <result> = FALSE

Example:

```
REPEAT: PRINT "Hello": UNTIL FALSE
```

FALSE returns the constant zero. It is used mnemonically in logical or conditional expressions.

HIMEM

Returns the top of memory.

Syntax: <result> = HIMEM

Example:

```
stacksize% = HIMEM - EXT
```

HIMEM returns the address of the top of memory used.

LOMEM

Returns the bottom of the heap.

Syntax: <result> = LOMEM

Example:

```
heapsize% = END - LOMEM
workspace% = LOMEM - TOP
```

LOMEM returns the address of the bottom of the heap.

PAGE

Returns the start of the program.

Syntax: <result> = PAGE

PAGE returns the address at which the object code starts.

TIME

Returns or alters the value of the centi-second clock. TIME may be used in two forms:

Syntax 1: <result> = TIME

Example:

```
PRINT TIME
```

TIME returns an integer giving the number of centi-seconds that have elapsed since the last time the clock was set to zero.

Syntax 2: TIME = <expression>

Example:

```
TIME = 0: REPEAT: UNTIL TIME > 500
```

TIME is followed by an expression yielding an integer value. It uses this value as a number of centi-seconds and sets the clock accordingly.

TIME$

Returns or alters the value of the real-time clock. TIME$ may be used in two different ways:

Syntax 1: TIME$

Example:

```
PRINT MID$(TIME$,INSTR(TIME$,".")+1)
```

TIME$ returns a 24 character string of the format:

```
Fri,23 Jun 1988.18:51:35
```

The date and time part are separated by a full stop '.'

Syntax 2: TIME$ = <expression>

Example:

```
TIME$="Fri,22 Jan 1988.20:11:45"
```

TIME$ is assigned the value of an expression which should be a string specifying the date, the time, or both.

The date must contain the first three letters of the day followed by a comma ',', then the number, the first three letters of the month and the year given in full. For example:

```
Mon, 13 Feb 1961
```

The time must contain the hours minutes and seconds as pairs of digits separated by colons ':'. For example:

```
04:35:00
```

If both are to specified the date should be given first and should be separated from the time by a full stop '.'.

TOP

Returns the top of the program.

Syntax: <result> = TOP

Examples:

```
workspace% = LOMEM - TOP
progsize% = TOP - PAGE
```

TOP returns the address of the end of the object code.

TRUE

Returns the logical value TRUE.

Syntax: <result> = TRUE

Example:

```
WHILE flag% = TRUE : PROCmainloop : ENDWHILE
```

TRUE returns the constant –1. It is used mnemonically in logical or conditional expressions.

19 : Miscellaneous Keywords

This chapter details all the 'odd' instructions which don't fall into one of the preceding categories.

DIM

Declares arrays or reserves a block of memory. DIM is used in two forms:

Syntax 1: DIM<identifier>(<expression>{[,<expression>}])

Examples:

```
DIM A%(10,10)
DIM char$(255)
```

DIM is followed by an identifier which can be any real, integer or string variable name. This should be followed by one or more integer expressions yielding non-negative values, separated by commas and enclosed in round brackets. DIM declares an array with the identifier given. It uses each value as the upper bound of a subscript, the lower bound is always zero.

It initialises numeric arrays to zeros and string arrays to null strings.

Syntax 2: DIM <variable><space><expression>

Example:

```
DIM bytes% size*10+1
```

DIM is followed by a variable which must be numeric, a space and an expression which yields an integer value. It uses this value as the number of bytes of memory required minus one and assigns the address of the first byte reserved to the variable.

It leaves the byte array uninitialised.

END

Terminates the execution of a program.

Syntax: END

END terminates the execution of a program. It is not always necessary in programs; execution will stop when the line at the end of the program is executed. However, END (or STOP) must be included if execution is to end at a point other than at the last program line. This prevents the definitions of procedures and functions from being executed.

LET

Assigns a value to a variable.

Syntax: LET <variable> = <expression>

Examples:

```
LET now% = TIME
LET a$(4) = CHR$(asc%)
LET mem%?offset%=x% + y%
```

LET must be followed by a variable of any type. It assigns the value of the expression on the right-hand side of the assignment to this variable. The expression must be of a suitable type, a numeric value cannot be assigned to a string and vice-versa.

The LET keyword is always optional in a normal assignment, but must not be used in the assignment to a pseudo-variable.

LINE$

This is a new function. It returns a string which is derived from the command used to start the program. This is mainly included for use in programs which are being compiled as modules.

The example module given in Chapter 20 demonstrates how it is used.

LOCAL

Declares a local variable in a procedure or function.

Syntax: LOCAL [<variable>] {[,<variable>]}

Example:

```
LOCAL a%, b, c$, d$
```

LOCAL may be followed by any number of variables of any type. It causes a new occurrence of each variable to come into existence. If the variable is numeric it will be initilialised to zero, if it is a string it will be initialised to a null string. These variables are in scope only inside the body of the procedure or function where they are declared, hence assignments to a local variable inside a procedure or function do not alter the global variable of that name, if one exists.

NB: The interpreter allows indirected expressions to be declared as local, for example:

```
LOCAL !&9000,!FNfred
```

This will make no sense under the compiler and will cause an error.

QUIT

Causes an exit from BASIC.

Syntax: QUIT

QUIT causes execution of the program to be terminated. The operating system command mode is entered.

REM

Indicates a remark.

Syntax: REM <string>

Example:

```
REM This procedure prints out a string in upper-case
```

REM can be followed by absolutely anything. It causes the rest of the line to be ignored. Hence, REM provides a means of commenting a program.

RUN

This will run a program

Syntax: RUN

Restarts a program from the beginning after initialising everything.

STOP

Terminates a program.

Syntax: STOP

Example:

```
IF x<0 PRINT "x= ";badval%:STOP
```

STOP causes the message 'Stopped' to be produced and terminates the execution of a program. END should be used as the normal means of ending a program, STOP is intended mainly as a debugging aid.

SWAP

Exchanges the value of two variables.

Syntax: SWAP <variable>,<variable>

Examples:

```
SWAP A%, B%
SWAP name1$, name2$
SWAP arr(1,2), arr(2,1)
```

SWAP must be followed by two variables which must already exist. They must either both be numeric or both be strings. It exchanges the contents of the two variables.

WAIT

Waits for start of the display frame.

Syntax: WAIT

WAIT halts execution of the program until the start of the next display frame. It enables a program to synchronise animation effects with the scanning of the display hardware.

20 : Relocatable Modules

When compiling a program ABC provides directives which allow the compiled program to be formatted into a relocatable module. There are three types of modules:

- Application
- Utility
- Service

Each of these is discussed below.

Application Modules

Programs which are compiled into an application module are run via a * command. The program will make use of all of the application workspace, or as much as may be specified using the STACK and HEAP compiler directives. When the program finishes, or an untrapped error occurs, it exits by executing OS_Exit (similar to the effect of QUIT from BASIC).

The module can be executed directly on loading via *RUN or *RMRUN. It can be re-executed using the *command which it provides.

Utility Modules

This is much like an application module with the exception that the program does *not* make use of the application workspace. Instead it uses memory claimed from the Relocatable Module Area. When the program finishes, or there is an untrapped error, it exits by executing OS_Exit (similar to the effect of QUIT from BASIC).

The module can be executed directly on loading via *RUN or *RMRUN. It can be re-executed using the *command which it provides.

Service Modules

Again this works in a similar fashion to an application module in that the program is called by a * command. As with Utility Modules the program will *not* make use of the application workspace, instead it uses memory claimed from the Relocatable Module Area. When the program finishes, or there is an untrapped error, it exits by returning to the calling program. Thus a service module provides a commandwhich can be called up from within another program.

The program which is incorporated within a service module is executed with the CPU in supervisor mode. This means that certain limitations are imposed:

Floating Point Operations

The Acorn Floating Point Emulator software cannot be called whilst in supervisor mode therefore any operation which involves floating point cannot be used.

This restriction rules out the use of floating point numbers,variables and arrays. The floating point indirection operator | cannot be used and the use of the EQUF directive in the assembler is not allowed.

In addition to the obvious floating point operations such as SIN, COS and TAN, there are a number of operations which, at first sight, appear to not involve floating point. For example the RND function makes use of the Floating Point Emulator and is therefore not usable within a service module.

The complete list of forbidden keywords is as follows:

ACS	ASN	ATN	COS
DEG	EXP	INT	LN
LOG	PI	RAD	RND
SIN	SQR	TAN	VAL

Note also that the ELLIPSE keyword may only be used in its four-parameter format. The fifth parameter is the angle of rotation and involves trigonometric calculations involving floating point numbers.

Finally, the use of READ and INPUT of integers may call up floating point operations if the data which is to be read is in floating-point form. The use of these keywords is not forbidden but this limitation must be observed.

Assembler Subroutines

Be very careful if your program includes any assembly language. In particular, you must ensure that assembler subroutine preserves the processor mode. Also, you should note that the use of SWI instructions when in supervisor mode causes Register 14 to be corrupted. Thus, a subroutine which calls a SWI must preserve R14 on the stack. Any such subroutine may not make use of the Floating Point Emulator.

Execution of the module directly via *RUN or *RMRUN will cause the module to be loaded and initialise. To execute the embedded program you must issue the *command which it provides.

Module Compiler Directives

A number of directives are provided to specify the details of the module.

Module Type

The type of module required must be specified using the MODULE TYPE directive thus:

```
REM {MODULE TYPE <type>}
```

The three possible types should be specified as follows:

```
REM {MODULE TYPE APPLICATION }
REM {MODULE TYPE UTILITY     }
REM {MODULE TYPE SERVICE     }
```

Title String

```
REM {MODULE TITLE <string>}
```

The title string of the module will be set to <string>.

Version String

The version string of the module will be set to <string>. This will be incorporated in the module help string as given in response to:

```
*HELP MODULES
```

The version string should be of the form x.yz

Command String

This is the command which is used to run the module.

```
REM {MODULE COMMAND <string>}
```

If the command string expects any arguments it should be followed by the number required.

```
REM {MODULE COMMAND <string> <args>}
```

If some of the arguments are optional then both the minimum and maximum number should be given.

```
REM {MODULE COMMAND <string> <minargs> <maxargs>}
```

Command Help String

The command should be provided with a help string. The text of the help string will begin with the command and then be followed by the text specified by this directive.

```
REM {MODULE HELP <text>}
```

Module Memory

For utility or service modules it is necessary to specify the amount of memory which will be claimed when the module is initialised.

```
REM {MODULE MEMORY = <size in bytes>}
```

An Example Module

The following example module illustrates the use of the various directives detailed above to produce a service module. The module provides the command "SCREEN" which must be followed by a parameter. This parameter is the textual name of the colour to which the screen background will be set.

```
 10   REM >$.example.module
 20
 30   REM {MODULE TITLE    DemoModule    }
 40   REM {MODULE VERSION 1.00           }
 50   REM {MODULE TYPE     SERVICE       }
 60   REM {MODULE COMMAND Screen    1    }
 70   REM {MODULE HELP  sets the screen background colour}
 80   REM {MODULE MEMORY = 1024 }
 90
100   I%=INSTR(LINE$," ")
110
120   Command$=LEFT$(LINE$,I%-1)
160
170   Command$=FNUPPER_case(Command$)
180
190   CASE Command$ OF
200   WHEN "RED"    :COLOUR 0,1
210   WHEN "YELLOW" :COLOUR 0,3
220   WHEN "GREEN"  :COLOUR 0,2
```

```
230   WHEN "BLUE"   :COLOUR 0,4
240   OTHERWISE
250   ERROR 1,"Bad Colour"
260   ENDCASE
270
280   END
290
300   DEF FNUPPER_case(A$)
310   LOCAL I%,B$,C$
320   FOR I%=1 TO LEN(A$)
330   C$=MID$(A$,I%,1)
340   IF "a"<=C$ AND C$<="z" THEN
350   C$=CHR$(ASC(C$)-32)
360   ENDIF
370   B$+=C$
380   NEXT
390   =B$
```

This module makes use of the LINE$ function to read the text of the command line which was used to start it. The Operating System will already have ensured that only one command line argument was present following the "screen" command as requested in the module specification on line 60. Thus all that is necessary is to skip to the first space, which is found using the INSTR function, take the rest of the string and convert to upper case.

The CASE statement then checks to see whether the argument is one which it recognises and performs the necessary COLOUR commands if it is. An error is generated if the argument was not one of the colours which the module provides. This error will be reported back to the program which issued the *SCREEN command.

If an incorrect number of parameters is given then the Operating System causes an error message to be given. This message is constructed by the complier from the information about the module and bound into it when the module is made. It will indicate what syntax the command will accept.

The module requires only a very small amount of workspace. In the example 1024 bytes are provided.

Once the module is complied and loaded the *SCREEN command can be used from another program. For example

```
10 REPEAT
20 INPUT colour$
30 OSCLI("SCREEN "+colour$)
40 UNTIL FALSE
```

Module Related Errors

FP ops not allowed in service code

This is given by the compiler if a floating point operation would be required within a program which is being compiled as a SERVICE module.

0 to 255 only

This error results from a MODULE COMMAND directive which is malformed or which requests a number of arguments outside the permitted range. The range is 0 to 255.

maxargs must be >= minargs

This will be given if the maximum number of command-line arguments is specified as lower than the minimum number. For example

```
REM {MODULE COMMAND 3 2}
```

Must be >= 0

The compiler will give this message in reply to a MODULE MEMORY directive which specifies a zero or negative size.

21 : Manifest Constants

It is generally much clearer to use named variables than to use explicit constants within a program. However there is a penalty to pay for this as program will be slower in operation and consume more memory than if the constant approach is used. To solve this problem the compiler supports the definition of manifest constants. That is it is possible to define a named value to represent a constant number throughout the program.

A manifest constant may be defined as follows:

```
DEF height% = 1023
```

Wherever height% is used within the program the value 1023 will be substituted instead. This helps to make programs much more readable. It is not possible to assign a new value to a manifest constant. This helps to make programs more secure against accidental alteration of "constants".

Similarly, it is possible to define string or floating-point constants.

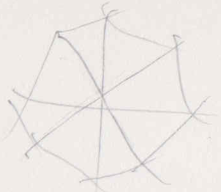
For example:

```
DEF text$ = "some text"
DEF approx_pi=3.2
```

A good technique is to use these in conjunction with the COMPILE and NOCOMPILE directives to set up a program which will run both under the interpreter and the compiler, whilst at the same time allowing the compiler to generate efficient code. For example:

```
DEF maximum% = 1000
DEF minimum% =  200
REM{NOCOMPILE}
maximum%=1000
minimum%= 200
REM{COMPILE}
```

This will make the interpreter set up and use the variables maximum% and minimum% whereas the compiler will use the two manifest constants. Such a program should run equally well under both systems but with the added security that the compiler will not allow the values to be changed.



22 : Conditional Compilation

Through the use of manifest constants it is possible to control whether the compiler looks at or ignores sections of the source program. This is best illustrated with an example.

```
 10 DEF DEBUG = 1
 20
 30
100 REM {IF DEBUG}
110 PRINT TAB(0,0)"Values are "I%+1,J%
120 REM {ELSE }
130 REM {ENDIF}
```

The manifest constant DEBUG is set to the value 1 on line 10. This will be taken as TRUE by the IF directive on line 100 and so the compiler will compile line 110. If DEBUG is set to 0, the IF directive will treat this as FALSE and line 110 will be ignored by the compiler. Source text between the ELSE and ENDIF directives is only compiled if the condition is FALSE.

Note that it is not possible to nest these compiler directives.

Dabhand Guides Guide

Archimedes Books

Archimedes Assembly Language

By Mike Ginns. Price £14.95. Spiral bound 368 pages.
ISBN 1-870336-20-8. Available now.
Programs disc £9.95 – £21.95 inclusive when ordered with book.

This is a complete guide to programming the Archimedes in machine code. Mike Ginns provides a clear, step-by-step account of using the assembler using simple but useful programs provide the practice to illustrate the theory thus making it ideal for the beginner. But this guide goes much further. For instance it explains how to use the Debugger and there is a large section on implementing BASIC equivalents in machine code plus coverage of Arthur and using SWIs, WIMPs and fonts.

The powerful Operating System is covered with details of how to access its many facilities from the machine code level. Within this are details of using graphics, sound, windows, the font painter and the mouse, all from within machine code programs. To make the transition from BASIC to machine code as painless as possible, the book contains a large section on implementing BASIC statements in machine code.

A programs disc accompanies the book which contains all of the various programs used in the text plus 11 extra utility programs including a disassembler, various memory manipulators and a disc sector editor. The programs disc is supplied with its own 16 page manual.

Reviews:

Risc User, July/August 1988: *"The style of the text throughout the book is easy to read...I would recommend Archimedes Assembly Language."*

Archive, August 1988: *"The actual explanations are lucid...Overall then, this is a comprehensive and wide ranging book which stands up well as both a tutorial on assembly language and as a guide to the programming environment and facilities provided by Arthur. I recommend it..."*

Archimedes Operating System

By Alex and Nick van Someren. Price £14.95. Spiral bound 250 pgs approx.
ISBN 1-870336-48-8. Available December 1988.
Programs disc £9.95 – £21.95 inclusive when ordered with book.

The ideal companion for *Archimedes Assembly Language* and *the* reference guide for all Archimedes user who take their computing seriously.

C is a standard language – programs the world over are written to this standard and in such a way as to allow them to be transferred to other machines and run again, in many cases with little or no editing required. A source program written in C on the Amstrad PC for instance would generally compile and run quite happily on the Archimedes, the Amiga, or any other PC for that matter.

Speed – a vital factor in the running of programs – is assured because a C program is compiled into ultra fast machine code. Write you very own commands in a friendly environment and let the C compiler transform it into machine code – no assembly language need be known! And what's more the original C source program remains intact for re-editing or fine tuning as you require.

Thirty-seven chapters, six appendices, a glossary and a comprehensive index make **C: A Dabhand Guide** probably *the* guide to programming in C. Included is a chapters on programming in C on the Archimedes, (and BBC and the Master 128/Master Compact for that matter).

Unique diagrams and illustrations help the reader to visualise programs and to think of in C. Assuming only a rudimentary knowledge of computing in a language such as BASIC or Pascal the reader is provided with a grounding in how to build up programs in a clear and efficient way.

To help the beginner a complete chapter on fault finding and debugging assists in tracing and correcting both simple and complex errors.

A Programs Disc is available for most of the major micros and this contains the listings in the book plus several other useful utilities including an adventure game and an indexer. The extra programs are documented in an informative manual.

The first review of **C: A Dabhand Guide** appeared in Beebug Magazine in June 1988 and it had this to say: "*The 512 pages cover all important aspects of C...the tone is friendly and the explanations are full and easy to understand without being patronising...the program structure diagrams which illustrate the larger programs are very helpful...the book being full of good advice about program design and layout. In conclusion, then, a very good, reasonably priced introduction to C for the non-specialist.*"

BASIC V

By Mike Williams. Price £9.95. Perfect bound 140 pages approx.
ISBN 1-870336-75-5. Available December 1988.

This book provides a practical guide to programming in BASIC V on the Acorn Archimedes. Assuming a familiarity with the BBC BASIC language in general it describes the many new commands offered by BASIC V, already acclaimed as one of the best and most structured versions of the language on any micro and is illustrated with a wealth of easy-to-follow examples throughout.

An essential aid for all Archimedes users it will also appeal to existing BASIC users who wish to be conversant with its many new features. BASIC V includes several new control structures which are major innovations. These are discussed and the text is littered with simple but effective examples. For the graphics programmer, the new extended graphics commands are covered with interesting examples of their use along with control and manipulation of the colour palette.

Other major topics covered include:

- WHILE, IF and CASE
- Use of mouse and pointer
- Local error handling.
- Sound
- The Assembler
- Matrix operations
- Functions and Procedures
- Operators and string handling
- Arthur
- Programming hints and tips

The author, Mike Williams, is Editor of Beebug magazine and Risc User, the largest circulation Archimedes specific magazine.

Archimedes First Steps

By Bruce Smith and Graham Stanley. Price £9.95. 100 pages approx.
ISBN 1-870336-73-9. Available December 1988.

Essential reading for all new and potential owners of the Archimedes. Assuming no prior knowledge of the micro it helps the reader through those first few hectic months of ownership of the world's fastest micro, from switching on to running third party software.

Written in a friendly informative style Bruce Smith and Graham Stanley explain how to use and get the most from the Desktop. The Welcome Disc contains a wealth of programs – these are documented and explained. The range of new * commands is perhaps the most baffling to new users and so the authors guide the reader through these and explain their actions.

Many owners will have existing BBC software and the transfer and use of this on the Archimedes is outlined, including the use of software such as VIEW. Add-on hardware and applications such as the disc drives, printers, the PC Emulator and the Archimedes Basic Compiler are also discussed.

The many features of this book include:

- The DeskTop & Welcome Disc
- The Basic Editor
- Configure and Setup
- Modules
- SWI Calls
- Using the View Family
- The PC Emulator
- Using ADFS
- Help Commands
- Converting programs
- The 6502 Emulator
- Using Arthur
- Hardware upgrades

Archimedes Software

Archimedes System Manager

By Mike Ginns. Price £49.95. ISBN 1-870336-77-1. Available Jan 1989.

The Archimedes System Manager is a powerful user interface that makes the handling of Archimedes information, files and system data an easy and simple task whatever your level of computing expertise. Written as a relocatable module ASM interfaces seamlessly with Arthur to provide a host of new facilities which are accessible as standard * commands. For beginners it will make those difficult and hazardous tasks second nature, while experienced users will find that the simplicity of ASM also makes it a most powerful management system.

ASM is supplied on disc and includes its own User Guide which contains full explanations on the use and applications of each facility. For more information on this product call for our complete specification sheet.

Archimedes Arcade Action!

Alerion

By Felix Andrew. Price £14.95. Available now.

"Alerion - an eagle without beak or feet is the Arcturian term for impossible and the codename for your mission!"

"Your space-fighter is equipped with revolutionary new equipment, not least a new radar cloaking system which renders you invisible, a holographic targeting system and un-limited fire power."

"To succeed you task is quite simple ... blow the living daylights out of anything that moves!"

Alerion is a welcome return to the most popular of computer games but utilising the power speed and superb graphics only available on the worlds fastest microcomputer – the Archimedes.

You have a bird's eye view of the action, your space-fighter flying over the varied enemy terrain, scrolling effortlessly beneath you. This is done by using the impressive 256 colour, high resolution mode. The game screen is refreshed 25 times a second by using two 80k screens in a highly innovative fashion, where one is dedicated to update while the other is used solely for display purposes, there is therefore no messy screen swapping.

As impressive is the use of digital sounds which were sampled from professional sources giving the game an authentic feel. Alerion – A mission impossible!

Arcendium

By David Acton and David Lawrence. Price £14.95. Available Nov 1988.

Arcendium provides four of the most popular board games on-disc for you to run and play on your Archimedes. Draughts, Reversi, Quadline (also called Four-in-a-Row) and Backgammon are the highly playable games displayed in the Archimedes high-resolution Mode 12 using its full 16 colour capability to full effect. Combined with 3D-effects, the end result is and addictive environment for games playing for all the family.

All three games have varying levels of play. Two players in any combination can play either against the micro or another player. When playing the Archimedes the level of play is controlled by setting the time that the computer is allowed to 'think' in steps from 0.1 to 60 seconds.

Index